Exceptional Interprocedural Control Flow Graphs for x86-64 Binaries[⋆]

Joshua A. Bockenek[1], Freek Verbeek[1,2], and Binoy Ravindran[1]

[1] Virginia Tech
[2] Open University of The Netherlands

Abstract. Standard control flow graphs (CFGs) extracted from binaries by state-of-the-art disassembly/decompilation tools do not include information about exception-related control flow. However, such information is useful when statically analyzing programs that utilize structured exceptions. To fill that gap, we propose the concept of *Exceptional Interprocedural Control Flow Graphs (EICFGs)*. These graphs extend traditional CFGs by adding edges for stack unwinding, frame cleanup, and try/catch behavior caused by thrown exceptions. We provide an approach for generating EICFGs from x86-64 binaries featuring C++ exceptions. The approach is based on symbolically executing an abstract semantics that includes binary-level exception-related function calls. We validated our abstract semantics by generating concrete test cases that were then evaluated using real binaries. We applied an implementation of our approach to 341 off-the-shelf x86-64 binaries compiled from C++ as well as C and Fortran source code. From those binaries, we identified 2574 unique throws and successfully resolved the exceptional control flow for every one of them. We show that resolving throws leads to increased instruction reachability and uncovers edges not found by state-of-the-art tools such as Ghidra.

Keywords: C++ exceptions, Binary Analysis, Control Flow Graphs

1 Introduction

Control flow graphs (CFGs) are key components in both compilation and analysis of software. They provide information on the runtime execution of a program, i.e., which control flow paths are dynamically possible. Typically, they are constructed during compilation using the source code as ground truth. In the field of binary analysis, however, they must be *reconstructed* from a binary. This is a challenge, since control flow transfers may be *indirect*, i.e., their jump target may be resolved

[⋆] This version of the contribution has been accepted for publication after peer review and shepherding but is not the Version of Record and does not reflect post-acceptance improvements or corrections. The Version of Record is available online at https://dx.doi.org/10.1007/978-3-031-64171-8_1. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use: https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

at runtime. This makes generating a CFG an undecidable problem, as the exact values of certain state parts can only be determined at runtime.

This problem is exacerbated when dealing with *exceptional control flow*, induced mainly by the C++ `throw` and `catch` statements. The target of a throw, i.e., to which instruction address the control flow is transferred after execution of a throw statement, is decided dynamically at runtime. It is based on, among other things, the current call/return stack, the current caught exception stack, and low-level information pertaining to rethrows and catch statements. Moreover, it inherently requires *interprocedural* analysis, as the function call history is relevant for assessing the throws' targets. Interprocedural analysis is challenging, as one cannot simply isolate a function and do analysis, but must consider the binary as a whole.

Existing state-of-the-art disassemblers/decompilers, such as IDA Pro, Ghidra, and Binary Ninja, do not provide sufficient information on exceptional control flow. Typically, they are able to extract exception information statically available in binaries (e.g., landing pad locations). They can even provide interprocedural control flow graphs. However, they do not perform the interprocedural static analysis required for reconstructing *exceptional* control flow. That is, they cannot trace the path from an exception throw site to the landing pad instructions that exception goes to in the process of unwinding. CFGs under such analyses will have *throw sites as terminating locations* with no outgoing edges.

This paper provides a tool for static, interprocedural, automated C++-exception-aware control flow analysis on the binary level. That tool produces Exceptional Interprocedural Control Flow Graphs (EICFGs), which document the direct exception-handling-related components of the state of the program, such as exception objects currently allocated, number of uncaught exceptions, and which exceptions are currently in a caught state (see Sections 2 and 3).

Effectively, EICFGs expose control flow edges not found by other work. The motivation behind exposing such edges is that it leads to more accurate reachability analysis: instructions that were deemed unreachable may now be considered reachable, or instructions that were considered reachable only through a single path may now be considered reachable through other paths as well. In general, we argue that improved binary-level reachability analysis is useful for security analysis, verification, and patching. Some more concrete examples of use-cases are:

- A disassembler/decompiler leverages reachability information to decide which parts of the binary are lifted. The information stored in EICFGs enable a decompiler to lift paths that were not found by existing tools.
- Binary-level verification, be it theorem proving, symbolic execution, or model-checking, typically aims to be overapproximative. This requires insight into *all* possible execution paths. An EICFG can aid such verification efforts.
- ROP-programming aims at finding executable sequences (gadgets) that end with an instruction that modifies control-flow. More insight into the possible execution paths of a binary increases may lead to a larger attack surface.

The tool itself targets stripped binaries using C++ exception handling that were compiled for the x86-64 instruction set architecture (ISA) and System V application binary interface (ABI). We assume that external calls do not produce exceptions themselves. We also do not model `setjmp`/`longjmp`.

A desired characteristic is that the produced EICFGs are *overapproximative*: every concrete path, i.e., every path possible during dynamic execution, is included in the EICFG. We informally – through test case generation – argue that the abstract semantics that are symbolically executed overapproximate the concrete semantics [2]. However, it may be the case that an indirection is unresolved, in which case not all paths are explored. In those cases, the CFGs are clearly annotated. We thus argue that EICFGs are overapproximative *modulo unresolved indirections*. To strengthen this claim, we validated our abstract transition rules for exception handling against the concrete implementations of the corresponding library functions (Section 4). This was done by generating abstract states, concretization to a concrete CPU state, running the function under validation in an instrumented binary, observing the CPU state, and verifying that the concrete transition is correctly contained in the abstract transition.

We applied the tool to 341 off-the-shelf x86-64 binaries compiled from C++, C, and Fortran source code (Section 5). The implemented tool was able to identify and trace 2574 unique throws. Dealing with exceptional control flow can increase average instruction reachability by 14 instructions per unique throw, with 188 average unwind edges in the possible unwinding paths from each throw. Those edges are ones tools such as Ghidra do not produce.

All code, scripts, and tested binaries are publicly available available at https://doi.org/10.5281/zenodo.11081942.

2 EICFGs

A normal CFG provides users with information on control flow transfers induced by *jumps* and *calls*. Given a specific jump or call instruction, one can look up in the CFG the set of successor instructions and a state predicate on which that successor selection is based. Such state predicates are typically represented by expressions over flags (for example, `CF` and `ZF` for the carry and zero flag) or a jump table calculation.

A summarized reproduction of the interprocedural control flow graph generated by Ghidra for an example program can be seen in Figure 2, generated from a binary compiled from the source code in Figure 1. In this example, function `main` can indirectly call both functions `foo` and `bar`. Each of these two can throw an exception, and if that happens, `main` can either rethrow that exception (the red box) or normally return (the green box).

While Ghidra can identify catch and cleanup landing pads (boxes LP in Figure 2), it cannot show that the throws will unwind there. IDA Pro and Binary Ninja produce similar results; they can identify landing pad locations intraprocedurally, but they do not trace exceptional control flow interprocedurally.

We have verified that this holds for more complicated programs as well (some of the programs used in Section 5).

In contrast, we produce an EICFG which augments the normal CFG with edges related to exceptional control flow (the dashed edges in Figure 2). These edges actually form a series of edges that lead from a `throw` to its landing pad, with labels containing predicates over, e.g., the current caught exception stack. Moreover, while not claiming it as a contribution in itself, we have taken care to resolve indirections whenever possible. The edge from `main` to `foo` and `bar` are due to indirect calls. Neither Ghidra, IDA-PRO or Binary Ninja generated these edges for this example.

```
int (*const FOOBAR[])(int) = {foo, bar};

int main(int argc, char* argv[]) {
  try {
    if (argc < 2)
      return FOOBAR[argc](argc);
  }
  catch (const std::exception&) {
    if (argc < 0)
      throw;
    else
      return 0;
  }
}
```

Fig. 1. Part of the C++ source code that was compiled to binary

Instead of simple labels, EICFGs use *exceptional state predicates* as labels. Informally, the information on which exceptional control flow is based is:

1. the exception object, e.g., type info and rethrown state;
2. the current set of return addresses on the stack;
3. the current uncaught exception count;
4. the current caught exception stack; and
5. a static address (landing pad) table for the unwinding process.

Exception type info is used to determine which catch blocks, if any, are applicable to the exception being thrown. Rethrown status is necessary when determining behavior when dealing with the binary equivalent of an argumentless `throw`. The return address stack is necessary to provide context for unwinding. The uncaught exception count keeps track of how many thrown exceptions are currently uncaught. This is useful for diagnostic information. Next, the caught exception stack provides information to set up implicit rethrowing. Finally, the landing pad table (LPT) maps from potential unwind spots in a binary to locations where unwinding can exit.

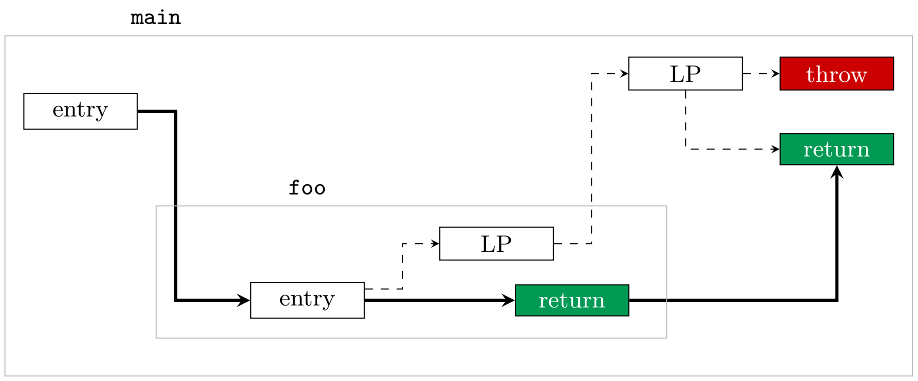

Fig. 2. Summarized view of a subpart of the control flow of Figure 1. Each box is itself a control flow graph over various instructions. The thick black edges are also generated by existing works, the dashed edges are not.

Definition 1. *An EICFG is a directed graph with instruction addresses as vertices and edges labelled with exceptional state predicates. There is an edge between instruction addresses a_0 and a_1 with label P if the instruction at address a_0 leads to instruction address a_1 for any state that satisfies predicate P.*

Some of the additional information provided by an EICFG is illustrated in Figure 3, which models the process of throwing an exception from the same example program as Figure 2. The representation in the figure indicates the process of unwinding from one throw site in the control flow graph to a try-catch block or cleanup landing pad. This path was triggered by the snippet of assembly shown in Figure 4, which allocates (0x125b), initializes (0x126d), and throws (0x1286) an exception. The landing pad table of the binary, LPT, directs the unwinding process: when stack unwinding reaches address i and $j \in \mathsf{LPT}(i)$, control flow branches to address j. Otherwise, another frame is popped off the stack. This will be elaborated on in Section 3.2. For this example, we get unwinding from address (0x1286) to 0x137e to 0x138b.

Due to overapproximation, there are two possible paths from that point. One is the path for an exception object that is not of the caught type, some checks of which occur via the assembly instructions 0x138f and 0x1393 of Listing 1.1. This path results in unwinding being resumed (0x1398) and leads to a bad termination case at instruction 0x116e. The other path continues from node 0x139d. It eventually leads to 0x116e from 0x13dd (intervening nodes elided). This is a good termination case as reaching that halting instruction occurred outside unwinding.

Resolving of indirections is shown in Figure 5. Whenever an indirection occurs, we detect whether the state is reached only via a conditional jump that positively bounds an index. If so, we concretize by all possible values for that index. In the given example, we detect values being read from a table with an index j strictly

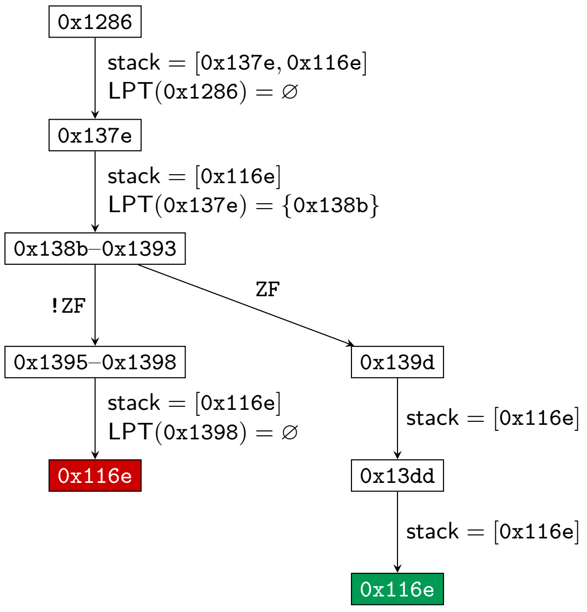

Fig. 3. EICFG fragment.

bounded by 2. This results in a computable indirection, and resolve an indirect call to either `foo` or `bar`.

3 Technical Formulation

The derivation of EICFGs from a binary requires four additional things: 1.) a static landing pad table, 2.) an abstract state model, 3.) an abstract transition relation, and 4.) a symbolic execution engine to apply the rules making up our abstract transition relation. We describe those here,using the following types: $\mathbb{B}$, $\mathbb{N}$, $\mathbb{Z}$, denote Boolean, natural and integral numbers respectively. $\mathbb{V}$ denotes *symbolic expressions*, which may be $\top$, indicating any or an unknown or undefined value. $\mathbb{P}$ denotes *immediate 64-bit addresses.* $\mathbb{R}$ denotes *registers.* $\mathbb{W}$ denotes *exception IDs*, and finally we have an enumeration $\mathbb{T} \coloneqq \{\top, \texttt{Good}, \texttt{Bad}\}$ for the program termination type.

3.1 Landing Pad Table

This information describes how unwinding should proceed given the unwinding reaching specific locations in a binary. It is extracted from the Common Information Entries (CIEs), Frame Description Entries (FDEs), and language-specific data areas (LSDAs) of the binary under test and assumed to be correct. In our current formulation, an entry in the catch table is merely a pointer to the

```
125b: call 10d0 <__cxa_allocate_exception>
1260: mov  rbx,rax
1263: lea  rsi,[rip+0xd9b] # 2005
126a: mov  rdi,rbx
126d: call 10e0 #std::domain_error init
1272: mov  rax,QWORD PTR [rip+0x2d4f]
1279: mov  rdx,rax
127c: lea  rsi,[rip+0x2abd] # 3d40
1283: mov  rdi,rbx
1286: call 1120 <__cxa_throw>
...
129c: call 10f0 <__cxa_free_exception>
```

Fig. 4. Assembly corresponding to Figure 3.

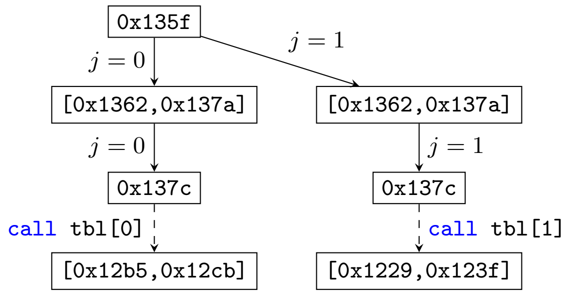

Fig. 5. Identifying indirection.

corresponding landing pad for this entry. While type pointers and exception specifications exist within the LSDAs as well, we do not currently utilize that information.

Thrown exceptions in C++ can be caught by catch blocks. Individual stack frames may require cleanup during the process of unwinding as well. The addresses of those catch blocks and cleanup routines are called landing pads. To accomplish reaching those addresses during unwinding, we require a *landing pad table*.

Definition 2. *A landing pad table* LPT *is a static map from instruction addresses to set of possible landing pads. Formally,* LPT*s have type* $\mathbb{P} \to 2^{\mathbb{P}}$.

Currently, we overapproximate and do not include exception type when determining landing pads.

Example 1. One of our running example landing pad entries is $\mathsf{LPT}(\texttt{0x137e}) = \{\texttt{0x138b}\}$. Thus, when an unwinding routine reaches instruction `0x137e`, that routine will jump to `0x138b`.

Exception Objects Exceptions of type $\mathbb{E}$ are records with named fields.

Listing 1.1. Example throw landing pad.

```
1387:    mov     eax, ebx
1389:    jmp     13d7
138f:    cmp     rdx, 0x1 # check for matching exception
1393:    je      139d
1395:    mov     rdi, rax
1398:    call    _Unwind_Resume
         # exception not handled by this catch block
139d:    mov     rdi, rax
13a0:    call    __cxa_begin_catch # start of catch block
13a5:    mov     QWORD PTR [rbp-0x18], rax
13a9:    cmp     DWORD PTR [rbp-0x24], 0x0
13ad:    jns     13b4
13af:    call    __cxa_rethrow # exception rethrow
13b4:    mov     ebx, 0x0
13b9:    call    __cxa_end_catch # end of catch block
13be:    jmp     1387
13c0:    endbr64
...
13d7:    # normally return
```

Definition 3.

$$\mathbb{E} := \begin{cases} \mathsf{rethrown} & : \mathbb{B} \\ \mathsf{handlerCount} & : \mathbb{Z} \end{cases}$$

This record has two fields. The Boolean field rethrown *indicates the rethrown status of the exception. Natural field* handlerCount *stores the current count of catch block handlers for the exception.*

3.2 Abstract State

Our exception-containing abstract states, type Σ, are records as well, with the following fields:

rmap: the register map This field has type $\mathbb{R} \to \mathbb{V}$. As previously stated, reading and writing registers smaller than 64 bits (e.g. `ebp` versus `rbp`) requires bit masking and shifting the underlying 64-bit register's value. This behavior is integrated into our symbolic execution engine. Larger registers (e.g. `xmm` and other vector registers) exist as operands in our instruction representation but are not used for state updates or reads.

stack: the return address stack Maintaining the current list of return addresses is necessary in order to perform stack unwinding and handle thrown exceptions. It is also helpful in detecting recursion. We make use of standard push/pop/peek functions. For the initial state, we have an empty stack. A `call` pushes the address of the next instruction to the stack, a `ret` pops. Some functions, such as `__libc_start_main` are given special treatment.

emap: the exception mapping This field has type $\mathbb{W} \to \mathbb{E}$. When an exception is created, it receives an ID based on its creation location and is stored in emap with that ID as the key.

terminated: the termination state This field has type $\mathbb{T}$. It defaults to value $\top$, indicating the program or function has not terminated yet. When a path of execution completes, it is set to either `Good` or `Bad`, indicating either normal or abnormal termination, respectively. We treat cases where an exception propagates to the top of the stack without being caught to be such "bad" cases.

Auxiliary exception variables Σ also contains a count of the number of currently-uncaught exceptions ($\mathsf{uncaught} \colon \mathbb{N}$) and a stack of the currently-caught exception IDs ($\mathsf{caught} \colon [\mathbb{W}]$). These fields are manipulated and used during entry to and exit from catch blocks as well as when rethrowing exceptions. This handling comes into play when dealing with nested catch blocks, exceptions (re)thrown within such blocks, etc.

Definition 4. *The type of abstract states, notation Σ, is a record with fields:*

$$\begin{array}{llllll} \mathsf{rmap} & : \mathbb{R} \to \mathbb{V} & \mathsf{stack} & : [\mathbb{P}] & \mathsf{emap} & : \mathbb{W} \to \mathbb{E} \\ \mathsf{terminated} & : \mathbb{T} & \mathsf{uncaught} & : \mathbb{N} & \mathsf{caught} & : [\mathbb{W}] \end{array}$$

To ease register references, for some state σ and named register $\mathtt{r}$, the notation $\sigma.\mathtt{r}$ is shorthand for $\sigma.\,\mathsf{rmap}(\mathtt{r})$, e.g. $\sigma.\mathtt{rdi} \equiv \sigma.\,\mathsf{rmap}(\mathtt{rdi})$.

3.3 Abstract Transition Rules

We define an abstract transition relation over abstract states. For various regular instructions we define semantics (e.g., `mov` and `add`). There is no need to provide semantics for the entire instruction set, as we can overapproximate values with $\top$ for irrelevant instructions. Conditional jumps are modeled non-deterministically. Unknown external functions are overapproximated by trashing the state according to a calling convention. Recursion is treated as follows: assume a call to a function inside the binary for some state σ. Then, if the return address to be pushed on the stack is already in $\sigma.\mathsf{stack}$, we instead treat that call as an unmodeled external call and continue execution past it.

Figure 6 provides a set of rules modeling exception-related ABI calls. The following abbreviations are utilized in those rules:

$$\begin{aligned}\text{handler}(id, \sigma) &\equiv \sigma.\,\mathsf{emap}(id).\mathsf{handlerCount}\\ \text{rethrown}(id, \sigma) &\equiv \sigma.\,\mathsf{emap}(id).\mathsf{rethrown}\\ \text{pushStack}(fr, \sigma, \sigma') &\equiv \sigma'.\mathsf{stack} = \text{push}(fr, \sigma.\mathsf{stack})\\ \text{popStack}(\sigma, \sigma') &\equiv \sigma'.\mathsf{stack} = \text{pop}(\sigma.\mathsf{stack})\\ \text{pushCaught}(c, \sigma, \sigma') &\equiv \sigma'.\mathsf{caught} = \text{push}(c, \sigma.\mathsf{caught})\\ \text{popCaught}(\sigma, \sigma') &\equiv \sigma'.\mathsf{caught} = \text{pop}(\sigma.\mathsf{caught})\end{aligned}$$

Notation for increment/etc.:

$$\begin{aligned}\text{handler}(id, \sigma')\text{++} &\equiv \text{handler}(id, \sigma') = \text{handler}(id, \sigma) + 1\\ \text{handler}(id, \sigma')\oplus &\equiv \text{handler}(id, \sigma') = |\text{handler}(id, \sigma)| + 1\end{aligned}$$

Notation $X{-}{-}$ follows the same concept, but the alternate for decrementing is $\text{handler}(id, \sigma')\ominus \equiv \text{handler}(id, \sigma') = -\,\text{sign}(\text{handler}(id, \sigma)) * (|\text{handler}(id, \sigma)| - 1)$. As a special case, $0\ominus = 0$.

Non-Unwinding Rules Figure 6a shows the rule for the special starting function that initiates the runtime used for standard features of C and C++, `__libc_start_main`. For this rule, we require the post-state's current instruction pointer be restricted to whatever was previously stored in `rdi`, `rcx`, or `r8`. We also require the stored return address to be on the top of a stack frame.

Next, Figure 6b illustrates the rule for function `__cxa_allocate_exception`. Our modeling assumes a system where virtual memory allocations always succeed (and the runtime terminates programs when they use up too much memory). It results in an exception object added to the exception map with the post-state σ''s instruction pointer as its ID. The object starts in a non-rethrown state and with no handlers. The ID is also set as the return value of the function in σ'.`rax`.

Example 2. After instruction `0x125b` of Figure 4, we have: σ'.`rax` $=$ `0x1260`, $\sigma'.\,\mathsf{emap}(\texttt{0x1260}).\mathsf{handlerCount} = 0$, and $\neg\sigma'.\,\mathsf{emap}(\texttt{0x1260}).\mathsf{rethrown}$.

The rule in Figure 6c is for the memory-related function `__cxa_free_exception`. This rule ensures the absence of an exception in the exception map based on the given ID. At our level of abstraction, the function `_Unwind_DeleteException` exhibits the same semantics and is thus elided.

Example 3. Consider instruction `0x129c` of Figure 4. The result of this instruction is $\sigma'.\,\mathsf{emap} = \varnothing$.

The rules in Figures 6d and 6e define `__cxa_begin_catch` behavior for different cases. For an exception not already caught, the associated rule pushes it onto the caught-exception stack. The rule for already-caught exceptions does not do this. However, both rules increment that exception's handler count and decrement the

$$\frac{\begin{array}{l}\sigma'.\texttt{rip} \in \sigma.\{\texttt{rdi}, \texttt{rcx}, \texttt{r8}\} \\ \mathrm{pushStack}(\sigma.\texttt{rip} + 5, \sigma, \sigma')\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(a) `__libc_start_main`

$$\frac{\begin{array}{ll}id = \sigma'.\texttt{rip} & \sigma'.\mathsf{emap}(id) = e \\ \sigma'.\texttt{rax} = id & \neg e.\mathsf{rethrown}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(b) `__cxa_allocate_exception`

$$\frac{\sigma'.\mathsf{emap}(\sigma.\texttt{rdi}) = \top}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(c) `__cxa_free_exception`

$$\frac{\begin{array}{ll} & \mathrm{handler}(id, \sigma')\oplus \\ id = \sigma.\texttt{rdi} & \mathrm{pushCaught}(id, \sigma, \sigma') \\ id \notin \sigma.\mathsf{caught} & \sigma'.\mathsf{uncaught}{-}{-}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(d) `__cxa_begin_catch` (not already caught)

$$\frac{\begin{array}{ll}id = \sigma.\texttt{rdi} & \mathrm{handler}(id, \sigma')\oplus \\ id \in \sigma.\mathsf{caught} & \sigma'.\mathsf{uncaught}{-}{-}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(e) `__cxa_begin_catch` (already caught)

$$\frac{\begin{array}{ll}id = \mathrm{peek}(\sigma.\mathsf{caught}) & \mathrm{handler}(id, \sigma') = 1 \\ \mathrm{handler}(id, \sigma) = 1 & \neg\,\mathrm{rethrown}(id, \sigma') \\ \mathrm{rethrown}(id, \sigma) & \mathrm{popCaught}(\sigma, \sigma')\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(f) `__cxa_end_catch` (have caughts, last handler, rethrown)

$$\frac{\begin{array}{ll}id = \mathrm{peek}(\sigma.\mathsf{caught}) & \mathrm{handler}(id, \sigma') = 1 \\ \mathrm{handler}(id, \sigma) = 1 & \sigma'.\mathsf{emap}(id) = \top \\ \neg\,\mathrm{rethrown}(id, \sigma) & \mathrm{popCaught}(\sigma, \sigma')\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(g) `__cxa_end_catch` (have caughts, last handler, not rethrown)

$$\frac{\begin{array}{l}\sigma.\mathsf{caught} = [] \\ \sigma'.\mathsf{terminated} = \mathsf{Bad}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(h) `__cxa_rethrow` alt

$$\frac{\begin{array}{l}\sigma \xRightarrow{\mathsf{U}^+} \sigma' \\ \sigma'.\texttt{rax} = \sigma.\texttt{rdi}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(i) `_Unwind_Resume` (LP(s))

$$\frac{\begin{array}{l}\sigma \xRightarrow{\mathsf{U}^-} \sigma' \\ \sigma'.\mathsf{terminated} = \mathsf{Bad}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(j) `_Unwind_Resume` (no LPs)

$$\frac{\begin{array}{l}\sigma \xRightarrow{\mathsf{U}^+} \sigma' \\ \sigma'.\mathsf{uncaught}{+}{+} \\ \sigma'.\texttt{rax} = \sigma.\texttt{rdi}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(k) `__cxa_throw` (LP(s))

$$\frac{\begin{array}{l}\sigma \xRightarrow{\mathsf{U}^-} \sigma' \\ \sigma'.\mathsf{terminated} = \mathsf{Bad}\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(l) `__cxa_throw` (no LPs)

$$\frac{\begin{array}{ll} & \sigma'.\mathsf{uncaught}{+}{+} \\ \sigma \xRightarrow{\mathsf{U}^+} \sigma' & \mathrm{handler}(id, \sigma')\ominus \\ & \mathrm{rethrown}(id, \sigma') \\ id = \mathrm{peek}(\sigma.\mathsf{caught}) & \sigma'.\texttt{rax} = id\end{array}}{\sigma \xrightarrow{\mathsf{A}} \sigma'}$$

(m) `__cxa_rethrow` (caught+LP(s))

Fig. 6. Abstract transition rules (unchanged state parts mostly elided).

state's count of uncaught exceptions. Though not listed, there is also a rule for an ID not currently in the exception map. That rule operates the same as our (elided) rule for unmodeled external calls. This allows for safe overapproximation.

Example 4. Consider instruction `0x13a0` of Listing 1.1. Assuming the existence of a valid exception object with ID *id* that was just thrown, the post-state σ' will satisfy: $\text{handler}(id, \sigma') = 1$, $\sigma'.\mathsf{caught} = [id]$, and $\sigma'.\mathsf{uncaught} = 0$.

The rules listed in Figures 6f and 6g then define some `__cxa_end_catch` behavior. The first rule applies when an exception ID is available on top of the caught stack, there are no more handlers for the corresponding exception object, and it is being rethrown. In this case, it is popped off the caught stack and no longer treated as being rethrown. The second rule applies when an exception is available, has no more handlers, and is not being rethrown. In that case, it is popped off the caught stack and removed from the exception map. Not shown is the rule for an exception that still has handlers remaining. In that case, its handler count is decremented but no other changes are made. Additionally, the case for an empty $\sigma.\mathsf{caught}$ again operates as an unmodeled external call for the sake of overapproximation.

Example 5. Consider instruction `0x13b9` of Listing 1.1. Assume the statements in Example 4 hold for the pre-state. Then, the post-state σ' for that instruction will satisfy $\sigma'.\mathsf{emap}(id) = \top$ and $\sigma'.\mathsf{caught} = []$.

Unwinding Rules In case of stack unwinding, the stack is recursively popped until one of two conditions occurs: a landing path is found or not. We respectively use $\xRightarrow{\mathsf{U}^+}$ and $\xRightarrow{\mathsf{U}^-}$ to indicate the compound stack unwinding transition from a state until one of those conditions is met.

Example 6. Assume $\sigma.\mathsf{stack} = [\texttt{0x116e}]$. Then, for $\sigma \xrightarrow{\mathsf{U}} \sigma'$ to hold, we must have $\sigma'.\mathsf{stack} = []$ and $\sigma'.\texttt{rip} = \texttt{0x116e}$.

Figures 6i and 6j show the simplest unwinding function rules, those for `_Unwind_Resume`. The main addition to the general unwinding transition is that, when landing pads are found, the original function argument ($\sigma.\texttt{rdi}$) is preserved in the result state's return register ($\sigma'.\texttt{rax}$). This models the concrete handling for carrying through exceptions during unwinding.

Example 7. Consider instruction `0x1398` of Listing 1.1. As previously described in Section 2, this instruction is intended to continue unwinding for exceptions that do not satisfy the source code's catch type specification. Assuming no more applicable landing pad table entries, the only valid post-states for the transition here match $\sigma'.\mathsf{stack} = []$ and $\sigma'.\mathsf{terminated} = \mathtt{Bad}$.

The rules for the initiating function `__cxa_throw`, shown in Figures 6k and 6l, expand on those for `_Unwind_Resume`. They add the condition that the post-state's uncaught exception count is incremented. As before, given our level of abstraction the function `_Unwind_RaiseException` is semantically equivalent to the base throw function and thus shares its rules.

Example 8. Consider instruction `0x1286` of Figure 4. We previously stepped through the process of throwing using this instruction in Section 2, so we merely state the results here. As this is the first throw at this time, we have $\sigma'.\mathsf{uncaught} = 1$. Additionally, the unwinding process stops for $\sigma'.\texttt{rip} \in \mathsf{LPT}(\texttt{0x137e}) = \{\texttt{0x138b}\}$, giving us $\sigma'.\texttt{rip} = \texttt{0x138b}$.

The rules for `__cxa_rethrow` in Figures 6h and 6m add a twist by utilizing the current caught-exception stack. When an exception object ID is available on the top of the caught stack, unwinding proceeds as usual. Furthermore, the corresponding exception object is marked as being rethrown and its ID is stored in `rax` for later usage. By contrast, when no caught exception objects are available, `__cxa_rethrow` must lead to an abnormal termination for strict modeling. However, that second rule can be relaxed for additional overapproximation by using the `_Unwind_Resume` rules instead.

Example 9. Consider instruction `0x13af` of Listing 1.1. Assume: $\sigma.\mathsf{caught} = [id]$, $\sigma.\,\mathsf{emap}(id) = e$, and $\mathsf{LPT}(\texttt{0x13af}) = \{\texttt{0x13c0}\}$. Then we end up with: $\sigma'.\texttt{rip} = \texttt{0x13c0}$ and $\sigma'.\,\mathsf{emap}(id).\mathsf{rethrown}$.

Additional rules exist for the process of *forced unwinding*, or manual stack unwinding. Those are summarized here, starting with `_Unwind_ForcedUnwind`. That function functions similarly to `__cxa_throw` (Figures 6k and 6l). However, instead of stopping based on landing pad table information, it executes the function stored in $\sigma.$`rsi` in each frame and uses the result to determine when to stop. `_Unwind_DeleteException` functions like `cxa_free_exception` (Figure 6c) at the end of that process. The helper func `_Unwind_GetIP` stores the current frame's instruction pointer in $\sigma'.$`rax`. Finally, the other helper function `_Unwind_GetRegionStart` stores the current procedure fragment's starting address in $\sigma'.$`rax`.

3.4 Symbolic Execution

We perform symbolic execution by application of the rules making up our abstract transition relation. The symbolic execution engine was developped in Haskell, specifically tailored for this purpose. For some initial abstract state σ_0, $\sigma_0.$`rip` is either manually provided or obtained from the binary's Executable and Linking Format (ELF) info. Then we iteratively fetch the instruction at that address, increment `rip` appropriately, and apply the applicable abstract transition rule to obtain successor states. If the transition rule results in multiple possible continuing states, we apply the symbolic execution step to each successor state. If no non-terminating states result, this path of execution ends.

To prevent infinite loops and alleviate some of the state space issues that can occur with such non-deterministic evaluation, we provide a join operation. This join operation is focused on exceptional state. From Σ it preserves emap, uncaught, and caught. To maintain contextual awareness, it also preserves `rip`, stack, and terminated. As an implementation detail, it also includes the temporary indices used by our jump table heuristic to ensure proper separation. All other state parts are combined, with priority given to the first equivalent state produced. For a more aggressive join, emap, uncaught, and caught can be excluded from the preserved state parts. The abstract transition rules are also simplified to support this exclusion.

3.5 Argument for Overapproximation

We consider a formal definition of the concrete transition rules our abstract ones overapproximate outside the scope of this paper. This is because our abstraction focuses on the domain of exceptional control flow in terms of its ABI-level definition. By contrast, concrete rules require a concrete implementation. Instead, we provide an informal argument for why our abstract transition rules are overapproximative.

First, for normal (non-exception-related) assembly instructions, our abstract transition rules default to assigning $\top$ to destination operands, overapproximating their effect. Only those instructions whose arguments affect exception- and stack-related behavior as well as global memory operations receive full modeling. They include `mov` and its relatives, `push`+`pop` and related instructions, and basic arithmetic/bitwise instructions. If we did not model those instructions, we could lose too much information concerning exceptional or even regular control flow.

Second, for exception-related function calls, the semantics in Figure 6 purposefully omit information from the abstract state. An example is the type of the exception being allocated. The abstract step function, then, considers *all* possible next states for *any* exception type.

With respect to indirections, we deploy a tactic for resolving them that, *if* successful, *then* it is overapproximative. Our tactic aims to find a program point (i.e., an instruction address) from which a variant of under-constrained and overapproximative symbolic execution [8] is executed from that program point up to the indirection. If the resulting symbolic state provides a finite bound to the set of jump targets of the indirection, the indirection is marked as resolved. Otherwise, it is deemed unresolvable.

In the case of unresolvable indirection, we do not apply additional heuristics or guesses. Instead, we stop further exploration at the indirection, if a jump, and clearly annotate the output accordingly. Unresolved indirect calls are treated as unmodeled external calls, but the same principle applies. We thus informally argue that the produced EICFG is overapproximative *modulo* unresolved indirections. If the EICFG is not annotated with any unresolved indirections, it is an overapproximation.

4 Validation

To increase trustworthiness, we validated some of our abstract transition rules against the corresponding real-world implementations. Specifically, we generated abstract states σ, and validated that:

$$\sigma \xrightarrow{\mathsf{A}} \sigma' \wedge \gamma(\sigma) \xrightarrow{\mathsf{C}} s' \implies \alpha(s') = \sigma' \tag{1}$$

Here α and γ denote abstraction and concretization functions (from the field of abstract interpretation [3]), and $\xrightarrow{\mathsf{C}}$ denotes concrete execution.

Abstract states σ are obtained through via *test case generation* [1]. For each rule under validation, we generated 10 000 arbitrary initial abstract states (σ) and then applied the rule to obtain the corresponding abstract post states (σ'). Then, using a test harness implemented as a combination of Python and the GNU Project debugger (GDB) scripts, we ran constructed real-world binaries featuring the desired concrete functions. The usage of GDB allowed easy interceding at specific points in the binaries in order to set up the initial state and extract the state after the step. Function γ operates before the concrete library function is executed, setting the state parts in Table 1 to their test case values. Function α operates after the concrete library function is executed, extracting the listed state parts from the concrete program state. The test harness then verifies that the abstracted state parts match the expected ones generated previously, satisfying Equation (1). Table 1 shows our validation status.

Table 1. Validated state parts.

Rule	`rip`	in/out regs	handlerCount	uncaught	handlerSwitchValue	caught
`__cxa_throw`	✓	✓	✓	✓		
`__cxa_begin_catch`	✓	✓	✓	✓	✓	
`__cxa_end_catch`	✓	N/A	✓	✓	✓	Partial
`__cxa_rethrow`	✓	✓	✓	✓		Partial
`_Unwind_Resume`		✓	✓	✓	✓	

Our constructed test programs are designed to be minimal but still call the specific library functions we provided abstract transition rules for. To easily read and write memory using GDB, we provided dummy versions of certain structs. Specifically, the hidden library structs for individual exception objects as well as global exception information. To utilize those structs within GDB, the programs must be built in debug mode.

4.1 Concretization and Abstraction

Next, the abstract start and end states are generated by a small wrapper around our abstract transition rules. We used components of the property testing library QuickCheck [1] to instrument the start state generation. The initial starting addresses are defined by the binary versions of the above-mentioned test programs. The end state generation is performed by applying a single step of our methodology to those test programs using those start states. The start states as well as the end states for each step are then exported for use by the test harness.

The concretization and abstraction functions γ and α are part of that test harness. As in abstract interpretation, they interface between the generated abstract states and the concrete memory layouts mentioned above. Both functions operate via GDB breakpoints that are set depending on the rule under test. Function γ operates before the concrete library function is executed, setting the state parts in Table 1 to their test case values. Function α operates after the concrete library function is executed, extracting the listed state parts from the concrete program state after the library function is executed. The test harness then verifies that the abstracted end state parts match the expected ones generated previously, satisfying Equation (1).

4.2 Observations and Challenges

During the process of this validation, we uncovered several implementation quirks that were not obvious. For example, the field handlerCount is actually a signed integer. This means that, when generating initial states, a negative value may be produced. As it turns out, the concrete implementation of `__cxa_begin_catch` takes the absolute value of negative handler counts supplied to it before incrementing that value. `__cxa_rethrow` performs a similar, but stranger, transformation. It decreases the magnitude by one, then inverts the sign; if the magnitude is 0, handlerCount is unchanged. The implementation of our abstract transition rules was updated to reflect those unearthed quirks.

We also did not cover those cases where an exception does not get caught and results in program termination due to stack unwinding. This is because we were unable to easily check the desired state parts in such cases. Similarly, we could not validate the `rip` modification of the function `_Unwind_Resume`. When running in GDB with our test program constructed to utilize `_Unwind_Resume`, handler switch value manipulation is required to trigger that path. That in itself is not necessarily an issue, but when that path is taken, control flow is ultimately redirected to the landing pad for the catch block that leads to the `_Unwind_Resume` rather than an appropriate parent landing pad. This prevents us from validating the target landing pad (or lack thereof) for that function (i.e. σ'.`rip`). However, we were still able to validate the other exceptional state components manipulated by that function.

5 Experimental Results

Here we present the results of generating EICFGs for 341 real-world programs and libraries. These programs and libraries have a variety of sizes and use cases and were sourced from places like GitHub and the Advanced Package Tool (APT) repositories. 49 of these programs utilize C++ exception handling (several despite the lack of exception handling tables) while all have been compiled for the x86-64 ISA and the System V ABI. The EICFG generation for each binary was executed on a server with four Intel® Xeon® E7-8890v4 CPUs (for a total of 96 cores) running at 2.20 GHz with 252 GiB of RAM. The server's OS was Ubuntu 18.04.5 LTS. Execution timeout was set to eight hours.

Ideally we would directly compare the number of edges found by our tool with the number of edges found by related work. That, however, does not give insight into what is gained by generating EICFGs. Existing tools, such as Ghidra, IDA Pro, and Binary Ninja produce fewer edges related to interprocedural exceptional control flow. However, they may produce more edges as they apply heuristics, pattern recognition, and best-guesses to resolve indirections.

We therefore directly compare the number of novel edges that are found using our approach versus an approach that does not consider exceptional control flow. This is a *baseline comparison*: a summary of the results comparing EICFG generation to the results obtained by running our tool with exception handling *disabled* (see Table 2). That version treats throw-related functions as terminating functions, while catch-related functions were treated as no-ops.

Table 2. Case study results.

Absolute Numbers								Baseline Comparison
Groups	Binaries	Instructions	Unwind Edges	Unique Throws	Caught Throws	Time (s)	Mem (GB)	Instruction Differential
NASA	13/14	1742K	1410	167	136	5056	36	1027
Xen	82/90	181K	0	0	0	112	1	0
Magick	15/17	172K	14	14	2	73	1	15
Cups	163/164	317K	3938	33	0	4049	2	0
Other	18/23	763K	63 260	830	526	5521	20	11 766
caf	5/6	632K	7952	466	254	918	12	4251
art	1/4	57K	216	31	30	151	10	1885
audio	5/7	100K	49 199	523	380	639	8	8509
drives	3/3	21K	21 683	69	61	273	6	674
games	3/7	35K	4081	260	254	55	1	3759
science	1/1	23K	3567	94	92	43	3	3009
tasking	1/1	23K	6	4	3	359	17	119
torrent	1/5	18K	328 948	83	76	4306	243	1195
	311/341	4089K	484 274	2574	1814	21 447	30	36 200

30 of the binaries we analyzed are not included in the table as the tool ran out of of memory or timed out due to state space explosion, in one case just in the baseline version. We identified 2574 unique throws and traced the exceptional control flow of each one. Based on our analysis, we were able to identify 760 of them as uncaught; the remaining 1814 all had a potential catch block in their unwinding path. On average, dealing with exceptional control flow can increase

coverage of reachable instructions by 14 instructions per unique throw, with each throw averaging 188 edges in the potential unwinding paths from that throw. Those edges are ones tools such as Ghidra do not produce.

The Xen binaries exhibited no change, as none contained any exceptional control flow. We have decided to cover some non-trivial binaries without exceptional behavior as well, to validate that our approach never decreases the number of edges found. Thus, other than the time it takes to run EICFG generation, there is no cost.

6 Related Work

Bottom-Up Approaches: Decompilers and Disassemblers. We provide a summary of works in Table 3. The table describes whether or not the works do intra- or interprocedural exception analysis and if so, if it is static or dynamic. Static approaches, such as the one presented in this paper, typically aim for overapproximation. They utilize abstraction or other methods to model paths symbolically. In contrast, dynamic approaches are inherently underapproximative. This is because they rely on concrete runtime behavior and evaluating all possible concrete paths is infeasible.

Table 3. Bottom-up Exceptional Analysis Comparisons

Program	Intraprocedural†	Interprocedural‡	URL
This work	**Statically**	**Statically**	
Binary Ninja	Statically	Dynamically	https://binary.ninja
IDA Pro	Statically	Dynamically	https://hex-rays.com/ida-pro/
Ghidra	Statically	Dynamically	https://ghidra-sre.org
McSema	Statically	No	https://github.com/lifting-bits/mcsema
RetDec	Unknown	No	https://github.com/avast/retdec

† Can it identify the landing pads in a function?
‡ Can it trace from (re)throw to landing pad?

While Binary Ninja, IDA Pro, and Ghidra all support some form of intraprocedural exception handling analysis, they can only perform interprocedural exception handling analysis dynamically via debugging. For example, Ghidra provides default, platform-dependent analyses that extract try-catch block information and other landing pad information for exception handling. However, it does not provide unwinding control flow in the generated block/call graphs, only cross-references for the `LPT`. This means that it can identify landing pads and analyze the code following them, but it cannot identify the exceptions that will reach them. Its built-in debugger that can perform unwinding is also ultimately just an interface to external, dynamic debugging tools such as GDB or LLDB, which perform dynamic analysis and instrumentation. Traces are also supported, but those require the program to have been run previously.

McSema is an executable lifter: It lifts machine code to LLVM bitcode. It provides intraprocedural exception analysis by generating the LLVM representation for exception landing pads. However, it does not perform interprocedural analysis as it merely lifts to LLVM bitcode, rather than tracing the execution of exceptions from throw site to landing pad. RetDec functions similarly to McSema as a decompiler-to-LLVM, though it also supports C output. However, we could not find information about its ability, or lack thereof, to deal with exception handling. As with McSema, it does not perform interprocedural exception analysis.

Top-Down Approaches. By contrast, there are tools that analyze exceptions from the source-code side [7,6,9]. This prevents the analysis of legacy code without source but allows for better static analysis during development, or even formal proofs of correctness of the exceptional semantics.

Hutton and Write [4] provided basic formal semantics for source-level C++-like exceptions. They accompanied this with a compiler for a small language with exceptions and a proof of correctness of that compilation. This is different from our approach as we do not aim to verify the correctness of exceptional behavior due to our lack of ground truth (source code or some program specification). Additionally, Prabhu et al. [7] provide source-level generation of interprocedural exception control-flow graphs (IECFGs) for C++ exceptions. These IECFGs are much like our EICFGs, but extracted from source code instead. Unlike our work, however, IECFGs are used to *eliminate* exceptions when compiling to a binary to make static analysis easier. They cannot be used to analyze already-compiled programs with exceptions. Jiang et al. [5] used the additional edges exposed by IECFGs to design test cases for path testing and branch testing that take exceptional behavior into account.

Zhang et al. provide semantics of exceptional control flow of C++ source code [9], and show how symbolic execution of their semantics can be used to discover exception handling bugs in real-life open-source software such as a JSON parser and an SQL server. They argue that the number of edges in a call graph increases by 22% when exception handling is taken into account, and note that the density of exception handling bugs is relatively high. This emphasizes the importance of properly dealing with exceptional control flow.

7 Conclusion

Many C++ programs exhibit exceptional control flow that standard CFG extraction tools in disassemblers and decompilers do not identify. To deal with that issue, we have provided EICFGs and a tool for generating them. EICFGs extend standard CFGs extracted from binaries by including nodes and edges for exceptional control flow. Our abstract transition relation for exceptional control flow has been informally – through testing – shown to overapproximate the concrete semantics, modulo unresolved indirections. Furthermore, we have applied our EICFG generator to 341 real-world programs and libraries. We identified 2574 unique throws and were able to trace each one's exceptional control flow: 1814 were potentially caught while 760 had no identified potential for being caught. On average, dealing with exceptional control flow can increase coverage of reachable instructions by 14 instructions per unique throw, with each throw averaging 188 edges in the potential unwinding paths from that throw. Those edges are ones tools such as Ghidra do not produce.

One of the main drawbacks of our work is a propensity for state space explosion. While we were able to target programs with over 400 000 instructions, our EICFGs generally do not scale far beyond that. Even for smaller programs, we experienced timeouts and out-of-memory cases when a significant number of control flow nodes and edges were generated. Methods of reducing the state space while maintaining interprocedural exceptional analysis would provide for increased scalability and the ability to target even larger programs. For example, modeling of exception type info and integrating it into the LPT determinations would allow pruning of dead branches, reducing the tool's overapproximation without introducing unsoundness.

Our approach is specific to C++. The main challenge in applying a similar analysis to binaries compiled from different languages with exception handling, is to define abstract transition rules for their binary-level implementations (see Figure 6). We argue that these semantics must be thoroughly tested, as small implementation details may have large impact on the soundness of the approach.

There are several potential client analyses for our work. These include (semi-) formal and verified decompilation, binary-level data-flow analysis tools, and binary-level security analysis tools that require knowledge about the reachability of instructions. These use cases align with the use cases of mature reverse engineering tools such as Ghidra, and we envision that this work will be integrated in such a tool suite.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Andrea Mambretti for their insightful comments and suggestions, which helped to greatly improve the paper. This work is supported by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028, by DARPA under Agreement No. HR00112090028, the US Office of Naval Research (ONR) under grants N00014-17-1-2297 and N00014-18-1-2665, NAVSEA/NEEC under grant N00174-16-C-0018, and US Naval Surface Warfare Center Dahlgren Division under grant N00174-20-1-0009.

References

1. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming. pp. 268–279 (2000)

2. Cousot, P.: Abstract interpretation. ACM Computing Surveys **28**(2), 324–328. https://doi.org/10.1145/234528.234740
3. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. pp. 238–252. https://doi.org/10.1145/512950.512973
4. Hutton, G., Wright, J.: Compiling exceptions correctly. In: Kozen, D. (ed.) Mathematics of Program Construction. pp. 211–227. No. 3125. https://doi.org/10.1007/978-3-540-27764-4_12
5. Jiang, S., Jiang, Y.: An analysis approach for testing exception handling programs. ACM SIGPLAN Notices **42**(4), 3–8 (2007)
6. Kechagia, M., Devroey, X., Panichella, A., Gousios, G., van Deursen, A.: Effective and efficient API misuse detection via exception propagation and search-based testing. In: Software Testing and Analysis. pp. 192–203. https://doi.org/10.1145/3293882.3330552
7. Prabhu, P., Maeda, N., Balakrishnan, G., Ivančić, F., Gupta, A.: Interprocedural exception analysis for C++. In: Mezini, M. (ed.) Object-Oriented Programming. pp. 583–608. No. 6813 in Lecture Notes in Programming and Software Engineering. https://doi.org/10.1007/978-3-642-22655-7_27
8. Ramos, D.A., Engler, D.: Under-constrained symbolic execution: Correctness checking for real code. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 49–64 (2015)
9. Zhang, H., Luo, J., Hu, M., Yan, J., Zhang, J., Qiu, Z.: Detecting exception handling bugs in c++ programs. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). pp. 1084–1095. IEEE (2023)