Abstract Interpretation for Exception-Aware Function Summaries in Binaries

Daniel Engel
Open University
Heerlen, The Netherlands
daniel.engel@ou.nl

Freek Verbeek
Open University
Heerlen, The Netherlands
freek.verbeek@ou.nl

Binoy Ravindran
Virginia Tech
Blacksburg, VA, United States
binoy@vt.edu

***Abstract*—Binary program analysis relies on accurate control-flow graphs (CFGs) to reason about program behavior. However, standard CFGs fail to capture exceptional control flow induced by C++ exception handling and runtime unwinding, making it difficult to precisely determine how functions may exit. We address this limitation by making exceptional control flow explicit at the binary level. Starting from ELF binaries and recovered control-flow graphs, we construct exceptional control-flow graphs (ECFGs) using exception-handling metadata to make implicit unwinding and handler-transfer edges explicit at the binary level. Based on this representation, we define a concrete semantics of exceptional execution and develop an abstract interpretation that computes function-level summaries of observable exit behavior, including normal returns, uncaught exceptions, and process termination. We implement the analysis and evaluate it on several large real-world C++ software systems. The results show that exception-aware summaries can be computed efficiently for binaries containing millions of instructions and capture a diverse range of exceptional behaviors, demonstrating the practicality of exception-aware abstract interpretation at the binary level.**

I. Introduction

Modern software systems rely heavily on precompiled libraries. When integrating such components, developers and analysis tools often need to determine whether a function returns normally, propagates an exception, or terminates execution. This information is critical for understanding observable program behavior in robust system design and binary analysis.

At the binary level, however, this behavior is difficult to recover. Standard control-flow graphs (CFGs) reconstructed from disassembly primarily capture ordinary control transfers and typically omit exceptional edges induced by stack unwinding and runtime termination mechanisms. As a result, exception propagation and termination behavior are often absent from analyses that rely on CFG structure alone, limiting their precision.

Consider a developer integrating a precompiled Boost library into a larger system. Even when documentation is available, it may not reflect the exact behavior of the deployed binary. For example, a function marked as `noexcept` is expected not to propagate exceptions. The compiler does not necessarily enforce this by preventing the function from throwing. Instead, its compiled implementation may still contain a reachable `throw`, and the compiler wraps the call site in a runtime-generated handler, roughly `try { f(); } catch (...) { terminate(); }`, that catches any escaping exception and terminates the program rather than letting it propagate.

Reasoning about exceptional behavior from source code alone is often insufficient. Source code may be unavailable or may not correspond to the deployed binary due to compiler optimizations, application binary interface (ABI) conventions, and runtime support mechanisms. As a result, the binary is the most faithful representation of executable behavior. This is particularly relevant for exception handling, where source-level constructs such as `throw` and `noexcept` do not directly correspond to observable control flow at the binary level. Consequently, precise analysis of exceptional behavior requires reasoning directly on binaries rather than source code.

In this work, we compute function-level exit-behavior summaries for binary programs. For each function, we determine the possible exit behaviors, including normal return, exception propagation, and process termination. These summaries characterize how a function may exit, independent of source-level annotations. The analysis is based on abstract interpretation over an exceptional control-flow graph (ECFG). The ECFG augments a standard control-flow graph with exception-related control transfers derived from binary exception metadata, including stack unwinding and runtime termination paths. This enriched representation serves as the substrate for computing exit-behavior summaries. This style of analysis follows classical abstract interpretation and interprocedural data-flow frameworks based on fixpoint computation and procedure summaries [3, 12].

We evaluated the analysis on real-world C++ binaries,

including libraries from the Boost ecosystem. The recovered exceptional control flow reveals behaviors that are not visible in ordinary CFGs and are difficult to infer reliably from documentation alone. As illustrative example, consider the function `regexecA` from the Boost regex library. The documentation describes it purely as an integer-returning function; the function is designed as a C/POSIX-style API that returns error codes rather than throwing exceptions[1]. The author notes in comments that the library as a whole may unlikely throw `std::bad_alloc`. However, our abstract interpretation over the binary-level implementation produces as function summary that `regexecA` may throw the exception `boost::wrapexcept<std::logic_error>`. Manual analysis confirms the existence of a path from function entry to this throw.

We make the following contributions:

- We define a semantics for exceptional control flow in binaries, capturing exception-related control transfers induced by stack unwinding.
- We develop an abstract interpretation that computes sound function-level exit-behavior summaries.
- We implement an analysis that automatically derives exceptional control-flow information and computes exit-behavior summaries for real-world binaries.

The remainder of this paper is structured as follows. Section II reviews related work on exception semantics in source-level and binary analysis settings, including prior approaches to exceptional control-flow recovery. Section III introduces the concrete model of exceptional control flow. Section IV presents the abstract interpretation used to compute exit-behavior summaries. Section V describes the implementation and evaluates the analysis on multiple real-world C++ projects. Section VI concludes the paper.

The implementation of the abstract interpretation is publicly available in the FoxDec repository[2].

II. Related Work

Research on exception handling has been studied in programming language semantics, compiler intermediate representations, and binary-level program analysis. We categorize existing work into three categories: (i) explicit modeling of exception-aware control-flow graphs for binaries, (ii) source-level semantics and analyses of exceptions, and (iii) control-flow reconstruction techniques for binary code.

Exception-aware control-flow graphs for binaries. The work most closely related to ours is the construction of exceptional interprocedural control-flow Graphs (EICFGs) [2]. While both approaches make exceptional control flow explicit in control-flow graphs recovered from binaries, they differ substantially in scope.

EICFGs model exceptional control flow across function boundaries by introducing explicit interprocedural edges that capture exception propagation throughout the program. The authors report that, while the approach can handle programs with over 400 000 instructions, it generally does not scale beyond this range due to state-space explosion. In contrast, our ECFGs are intraprocedural and serve as the basis for an abstract interpretation that computes summaries of function exit behavior. Interprocedural reasoning is performed through these summaries rather than through explicit cross-function control-flow edges, which yields a more compact representation and allows us to scale to binaries with several million instructions.

Despite this difference in scope, both approaches share the central idea of making exceptional control flow explicit at the binary level. Our work extends this line of research from exception-aware control-flow reconstruction to abstract interpretation over exception-aware control-flow graphs.

Source-level semantics of exceptions. A substantial body of work studies exception handling at the source-code level, ranging from formal semantics and exception-aware compilation schemes [5, 6] to verification and model checking of C++ exception behavior [14, 11]. Several works exploit interprocedural exception control-flow graphs (IECFGs) for program analysis tasks such as API misuse detection and bug finding [10, 7, 16]. Unlike these approaches, our work operates directly on binaries, where source-level exception structure is not available.

Control-flow reconstruction for binaries. Control-flow reconstruction from machine code is typically formulated as an abstract interpretation over instructions to recover basic blocks and control-flow edges [9, 8, 1]. Binary analysis frameworks such as FoxDec [15] and angr [13] implement this at scale, but focus on standard control transfers and typically omit exceptional control flow due to its complexity. Decompilers such as SmartDec [4] and frameworks including RetDec[3], Ghidra[4], IDA Pro[5], Binary Ninja[6], and Radare2[7] do recover exception handling constructs, but only intraprocedurally; resolving interprocedural exception propagation generally requires dynamic execution of the binary. This motivates our approach of extending reconstructed CFGs with explicit, statically derived exception-aware control-flow information.

III. Concrete ECFGs

While full control-flow graphs capture all possible execution paths, most of this information is irrelevant for reasoning

[1] https://www.boost.org/doc/libs/latest/libs/regex/doc/html/boost_regex/ref/posix.html
[2] https://github.com/ssrg-vt/FoxDec
[3] https://github.com/avast/retdec
[4] http://ghidra.net
[5] https://hex-rays.com/ida-pro
[6] https://binary.ninja
[7] https://rada.re

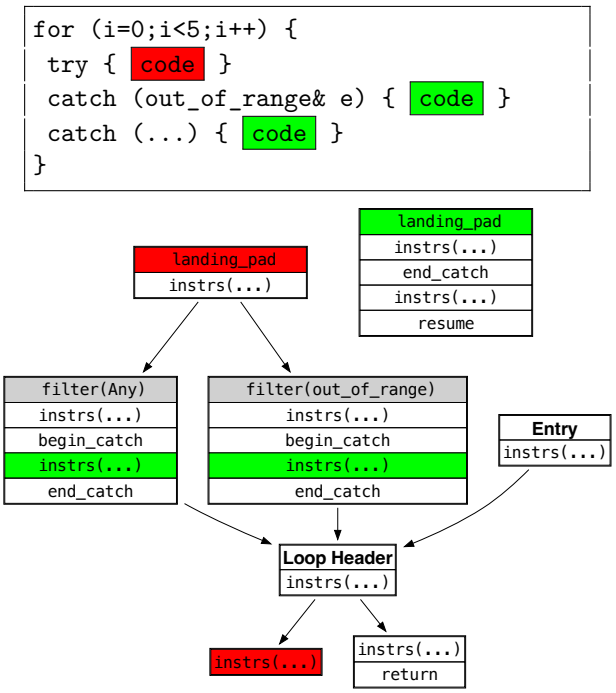

Fig. 1: Example of an ECFG for a binary. Edges with a single predecessor and successor are collapsed into the table rows; only edges representing actual control-flow branching are drawn explicitly.

about exception behavior. Normal control flow does not affect exception propagation or exceptional exits, and retaining it increases graph size and analysis cost. We therefore abstract away non-exceptional structure and focus only on exception-relevant control flow.

An ECFG captures all binary-level exceptional behavior of a function while eliding all "normal" control flow. This makes it a very compact representation of the binary-level behavior of a function. Still, an ECFG is fully executable, i.e., it can be given a well-defined semantics.

Figure 1 provides an example. The top shows C++ source code containing a `try-catch` construct inside a loop, where each block may contain arbitrary ordinary control flow. The corresponding ECFG is significantly more compact, as it abstracts away most ordinary control flow and focuses on exceptional behavior. In particular, all instructions are collapsed into opaque `instr` blocks, and loop structure is not explicitly represented since it is irrelevant for exception modeling. Instead, exceptional control flow is highlighted using colored regions: a red region for the `try` block and a green region covering both `catch` blocks and their associated landing pad. Ordinary CFG edges are shown as usual, while exceptional edges are encoded by matching colored regions to landing pads of the same color. Notably, while the red region directly corresponds to the source-level `try` block and its landing pad handles exceptions thrown within it by transferring control into one of the `catch` blocks, the green region does not correspond to any explicit source construct but instead represents compiler-generated cleanup code whose landing pad resumes unwinding when an exception is thrown inside the `catch` blocks.

We define a concrete semantics for ECFGs to reason about exception handling in binaries. The semantics abstracts away instructions that do not affect the call or exception stack by collapsing them into opaque state updates. This allows us to focus on control flow, calls, and exception propagation while treating the remainder of the machine state abstractly. The following formalization defines concrete states and notation used in the small-step semantics.

A. Syntax

We assume a set of exception types τ with a subtyping relation $\leq$, used by `filter` nodes to select matching handlers. A distinguished top type `Any` matches all exceptions.

Definition 1 (Syntax of ECFG Nodes)**.** *The set of ECFG nodes, denoted $\mathcal{N}$, is defined by*

$$\begin{aligned} n \in \mathcal{N} ::= \ & \texttt{instrs}(i_1, \ldots, i_n) \\ & \mid \texttt{call}(f) \mid \texttt{return} \mid \texttt{terminate} \\ & \mid \texttt{throw}(\epsilon) \mid \texttt{rethrow} \mid \texttt{resume} \\ & \mid \texttt{landing\_pad} \mid \texttt{filter}(\tau, T) \\ & \mid \texttt{begin\_catch} \mid \texttt{end\_catch} \end{aligned}$$

Clarifying remarks. The `instrs` nodes serve a role similar to basic blocks. They may contain jumps and loops, but calls are excluded since they modify the call stack. Indirect calls (e.g., a call through a register such as `call %rax`) are not represented by a separate node type. Instead, all call instructions are mapped to the same node $\texttt{call}(f)$. The $\texttt{throw}(\epsilon)$ node throws a specific exception ϵ. The `rethrow` and `resume` nodes instead operate on the exception currently being propagated: they are similar operations but apply in different contexts, as `rethrow` only works on the exception already being handled, while `resume` applies to the exception not yet being handled. The `landing_pad` and `filter` nodes are generated from the `.gcc_except_table` section of the binary. A $\texttt{filter}(\tau, T)$ node is executed only if the type of the current top exception is a subtype of τ and no other type in T provides a better match. Here, T is the set of types of all other `filter` nodes with the same predecessor. The type information for exception filtering is also extracted from the `.gcc_except_table`. The `end_catch` node may operate on either the top or second-to-top exception on the stack; the precise behavior will be formalized in the small-step semantics.

Definition 2 (Exceptional CFG (ECFG))**.** *An* exceptional control-flow graph *is a tuple*

$$G_e = \langle N, E \cup E_{\text{exc}}, n_0 \rangle$$

where $N \subseteq \mathcal{N}$ is a set of nodes, $E \subseteq N \times N$ are the standard CFG edges, $E_{\mathsf{exc}} \subseteq N \times N$ are additional edges representing exception handling (edges to landing-pads), and $n_0 \in N$ is the entry node.

ECFG are constructed in such a way that exactly the landing pads are reachable via exceptional edges

$$\langle n_1, n_2 \rangle \in E_{\mathsf{exc}} \implies n_2 = \texttt{landing\_pad}$$
$$\langle n_1, n_2 \rangle \in E \implies n_2 \neq \texttt{landing\_pad}$$

B. Semantics

A *concrete state* captures the runtime information relevant to exception handling and function calls. It consists of the machine state, a call stack, and an exception stack.

Definition 3 (Concrete Exception Item)**.** *An* exception item $x \in Ex$ *has the form*

$$x = \langle m, \epsilon, h \rangle$$

where $m \in \{\circlearrowleft, \triangledown, \Cap\}$ represents the mode of the exception (Unwinding, Landed, Handling), ϵ is the actual exception value (e.g., an instance of a language-level exception such as `std::exception`*), and $h \in \mathbb{N}$ is the handler count for this exception.*

Definition 4 (Concrete Exception Stack)**.** *An* exception stack $X \in ExStack$ *is an ordered list of exception items of the form*

$$X = [x_1, \ldots, x_n].$$

Definition 5 (Concrete State)**.** *The set of concrete states is denoted by $\mathcal{C}$. Each concrete state $c \in \mathcal{C}$ has the form*

$$c = \langle \mu, \Gamma, X \rangle$$

or it is the special `TERMINATED` *state. Here μ denotes the state of the machine, i.e., the contents of registers, memory and flags, $\Gamma = [n_1, \ldots, n_k]$ is an ordered list of CFG nodes and represents the call stack, and X is the exception stack.*

Definition 6 (`end_catch` handler)**.** *We define the operation* end-handle $: ExStack \rightharpoonup ExStack \times N$ *which implements decrementing the handler count and removal of the correct exception, and selection of the correct destructor. It will be invoked by the semantics of the* `end_catch`*.*

$$(a)\quad \mathsf{end\text{-}handle}(\langle \triangledown, \epsilon_1, h_1 \rangle \cdot \langle \Cap, \epsilon_2, 1 \rangle \cdot X) = \langle \langle \triangledown, \epsilon_1, h_1 \rangle \cdot X, \epsilon_2.\texttt{dtor} \rangle$$
$$(b)\quad \mathsf{end\text{-}handle}(\langle \triangledown, \epsilon_1, h_1 \rangle \cdot \langle \Cap, \epsilon_2, h_2 \rangle \cdot X) = \langle \langle \triangledown, \epsilon_1, h_1 \rangle \cdot \langle \Cap, \epsilon_2, h_2 - 1 \rangle \cdot X, \texttt{NOP} \rangle$$
$$(c)\quad \mathsf{end\text{-}handle}(\langle \Cap, \epsilon_1, 1 \rangle \cdot X) = \langle X, \epsilon_1.\texttt{dtor} \rangle$$
$$(d)\quad \mathsf{end\text{-}handle}(\langle \Cap, \epsilon_1, h_1 \rangle \cdot X) = \langle \langle \Cap, \epsilon_1, h_1 - 1 \rangle \cdot X, \texttt{NOP} \rangle$$

where $\epsilon.$`dtor` *is the entry of the exception's destructor function and the special* `NOP` *function does nothing.*

Equations (a) and (b) handle the special case where an `end_catch` *for the previous exception is reached while the* `begin_catch` *for the current exception has not yet been reached (this is valid behavior). Equations (a) and (c) handle the cases where the handler count is decremented to 0 and the exception needs to be destroyed. Equation (d) is the standard case, where the handler count of the top exception is decremented without destroying anything.*

The semantics of an ECFG is given as a transition relation over nodes and concrete states. A transition describes how the execution of a node affects the machine state, the call stack, and the exception stack, and how control flows along the edges of the graph. Formally, for a given ECFG $G_e = (N, E \cup E_{\mathsf{exc}}, n_0)$, we define the small-step relation

$$(\rightarrow) \subseteq (N \times \mathcal{C}) \times (N \times \mathcal{C})$$

The individual transition rules defining $(\rightarrow)$ are given in Figure 2. We assume a relation $\mu \rightsquigarrow_i \mu'$ to exists that captures the possible outcomes of executing the instructions of an `instrs`-node. Each rule specifies how a particular kind of node transforms the concrete state and determines the successor node along an outgoing edge of the ECFG. Standard control-flow edges $(n, n') \in E$ are taken during normal execution, while exceptional edges $(n, n') \in E_{\mathsf{exc}}$ and unwinding steps are taken when an exception is being propagated. In particular, exception propagation is driven by the exception stack, whose top element determines whether execution proceeds normally or continues unwinding. For unwinding across call-frames, we assume the frame restoration function $\mathsf{restore} : N \times Machine \to Machine$ to exist. This function corresponds to call frame information (CFI) directives in the program's assembly code.

The rules in Figure 2 separate normal execution and exception handling. Standard control flow is modeled by Instr, Call, Return, and Terminate. Throw initiates unwinding, while Rethrow, Resume, and Landing update exception modes during propagation. Filter, Begin-Catch, and End-Catch implement handler selection and lifecycle management, including destructor invocation. Unwinding proceeds either along exceptional edges (Unwind-1) or by restoring call frames (Unwind-2).

The semantics of a program is given by the reflexive transitive closure $\rightarrow^*$ of the transition relation. A program execution is a (finite or infinite) sequence

$$(n_0, c_0) \rightarrow (n_1, c_1) \rightarrow \cdots$$

where n_0 is the entry node of the program's ECFG and c_0 is some initial state, such that each step follows the rules in Figure 2.

IV. Abstract ECFGs

In this section, we develop an abstract interpretation of ECFGs for reasoning about function exit behavior. We abstract exception stacks, define an intraprocedural

$$\textsc{Instr}\ \frac{n_1 = \texttt{instrs}(i_1, \ldots, i_n) \qquad \langle n_1, n_2\rangle \in E \qquad \mu \rightsquigarrow_{i_1} \cdots \rightsquigarrow_{i_n} \mu' \qquad X \neq \langle \circlearrowleft, \epsilon, h\rangle \cdot X'}{\langle n_1, \langle \mu, \Gamma, X\rangle\rangle \to \langle n_2, \langle \mu', \Gamma, X\rangle\rangle}$$

$$\textsc{Call}\ \frac{n_1 = \texttt{call}(f) \qquad \langle n_1, n_2\rangle \in E \qquad X \neq \langle \circlearrowleft, \epsilon, h\rangle \cdot X'}{\langle n_1, \langle \mu, \Gamma, X\rangle\rangle \to \langle f, \langle \mu, n_2 \cdot \Gamma, X\rangle\rangle} \qquad \textsc{Return}\ \frac{n_1 = \texttt{return} \qquad X \neq \langle \circlearrowleft, \epsilon, h\rangle \cdot X'}{\langle n_1, \langle \mu, n_2 \cdot \Gamma, X\rangle\rangle \to \langle n_2, \langle \mu, \Gamma, X\rangle\rangle}$$

$$\textsc{Terminate}\ \frac{n = \texttt{terminate} \qquad X \neq \langle \circlearrowleft, \epsilon, h\rangle \cdot X'}{\langle n, \langle \mu, \Gamma, X\rangle\rangle \to \langle n, \texttt{TERMINATED}\rangle} \qquad \textsc{Throw}\ \frac{n = \texttt{throw}(\epsilon) \qquad m \in \{\circlearrowleft, \triangledown\} \qquad X \neq \langle m, \epsilon', h\rangle \cdot X'}{\langle n, \langle \mu, \Gamma, X\rangle\rangle \to \langle n, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, 0\rangle \cdot X\rangle\rangle}$$

$$\textsc{Rethrow}\ \frac{n = \texttt{rethrow}}{\langle n, \langle \mu, \Gamma, \langle \Uparrow, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle} \qquad \textsc{Resume}\ \frac{n = \texttt{resume}}{\langle n, \langle \mu, \Gamma, \langle \triangledown, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle}$$

$$\textsc{Landing}\ \frac{n_1 = \texttt{landing\_pad} \qquad \langle n_1, n_2\rangle \in E}{\langle n_1, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n_2, \langle \mu, \Gamma, \langle \triangledown, \epsilon, h\rangle \cdot X\rangle\rangle}$$

$$\textsc{Filter}\ \frac{n_1 = \texttt{filter}(\tau, T) \qquad \langle n_1, n_2\rangle \in E \qquad \epsilon : \tau_\epsilon \qquad \tau = \mathsf{min}\{t \in T | \tau_\epsilon \leq t\}}{\langle n_1, \langle \mu, \Gamma, \langle \triangledown, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n_2, \langle \mu, \Gamma, \langle \triangledown, \epsilon, h\rangle \cdot X\rangle\rangle}$$

$$\textsc{Begin-Catch}\ \frac{n_1 = \texttt{begin\_catch} \qquad \langle n_1, n_2\rangle \in E}{\langle n_1, \langle \mu, \Gamma, \langle \triangledown, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n_2, \langle \mu, \Gamma, \langle \Uparrow, \epsilon, h+1\rangle \cdot X\rangle\rangle}$$

$$\textsc{End-Catch}\ \frac{n_1 = \texttt{end\_catch} \qquad \langle n_1, n_2\rangle \in E \qquad \langle X', dtor\rangle = \mathsf{end\text{-}handle}(X) \qquad \langle dtor, \langle \mu, n_2 \cdot \Gamma, X'\rangle\rangle \to \langle n_2, \langle \mu', \Gamma, X'\rangle\rangle}{\langle n_1, \langle \mu, \Gamma, X\rangle\rangle \to \langle n_2, \langle \mu', \Gamma, X'\rangle\rangle}$$

$$\textsc{Unwind-1}\ \frac{\langle n_1, n_2\rangle \in E_{\mathsf{exc}} \qquad n_2 = \texttt{landing\_pad}}{\langle n_1, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n_2, \langle \mu, \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle}$$

$$\textsc{Unwind-2}\ \frac{\nexists n_2.\langle n_1, n_2\rangle \in E_{\mathsf{exc}} \qquad \mu' = \mathsf{restore}(n_1, \mu)}{\langle n_1, \langle \mu, n_3 \cdot \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle \to \langle n_3, \langle \mu', \Gamma, \langle \circlearrowleft, \epsilon, h\rangle \cdot X\rangle\rangle}$$

Fig. 2: Inference rules for the concrete small-step semantics.

analysis over ECFGs, and extract summaries describing possible exits such as returns, throws, and termination. Correctness is established via a Galois connection and a simulation argument.

A. Abstract Domain

To analyze function exit behavior, we abstract exception stacks by replacing concrete handler counts with bounded *abstract counts*.

Definition 7 (Abstract Handler Count)**.** *An* abstract handler count $\hat{h} \in \hat{H}$ *ranges over the set*

$$\hat{h} \in \{0, 1, \ldots, \hat{h}_{\mathsf{max}}, \top\}$$

Intuitively, $\hat{h}$ *represents an imprecise handler count where* $\top$ *denotes overflow over the maximum value* $\hat{h}_{\mathsf{max}}$*.*

Arithmetic on the bounded handler count follows the standard definition

$$\hat{h} \oplus 1 \triangleq \begin{cases} \top & \text{if } \hat{h} = \top \\ \top & \text{if } \hat{h} = \hat{h}_{\mathsf{max}} \\ \hat{h}+1 & \text{else} \end{cases} \qquad \hat{h} \ominus 1 \triangleq \begin{cases} \top & \text{if } \hat{h} = \top \\ 0 & \text{if } \hat{h} = 0 \\ \hat{h}-1 & \text{else} \end{cases}$$

In Section V, we use $\hat{h}_{\mathsf{max}} = 2$; the $\top$ value is never reached.

Definition 8 (Abstract Exception Item)**.** *An* abstract exception item $\hat{x} \in \hat{Ex}$ *has the form*

$$\hat{x} = \langle m, \tau, \hat{h}\rangle$$

where $m \in \{\circlearrowleft, \triangledown, \Uparrow\}$ *is the same mode as in Definition 3,* τ *is the exception type, and* $\hat{h}$ *is the abstract handler count.*

Definition 9 (Abstract Exception Stack)**.** *An* abstract exception stack $\hat{X} \in \widehat{ExStack}$ *is an ordered list of abstract exception items*

$$\hat{X} = [\hat{x}_1, \ldots, \hat{x}_n].$$

Definition 10 (Abstract State)**.** *An abstract state* $a \in \mathcal{A}$ *consists solely of an abstract exception stack*

$$\mathcal{A} \triangleq \widehat{ExStack}.$$

We now lift these sets of states to domains suitable for abstract interpretation. The concrete domain consists of sets of concrete states, and the abstract domain consists of sets of abstract states.

Definition 11 (Concrete and Abstract Domains)**.** *The* concrete domain D *is the powerset of concrete states, ordered by subset inclusion and the* abstract domain $D^\sharp$ *is*

the powerset of abstract states, plus a top element, similarly ordered by subset inclusion:

$$D \triangleq \mathcal{P}(\mathcal{C}), \quad d_1 \sqsubseteq_C d_2 \Leftrightarrow d_1 \subseteq d_2,$$
$$D^\sharp \triangleq \mathcal{P}(\mathcal{A}) \cup \{\top\}, \quad d_1^\sharp \sqsubseteq_A d_2^\sharp \Leftrightarrow d_1^\sharp \subseteq d_2^\sharp,$$
$$\text{where } d^\sharp \sqsubseteq_A \top, \text{ for all } d^\sharp$$

The empty set serves as a natural bottom element for the abstract domain.

B. Intraprocedual Analysis

Having defined the abstract domain of exception stacks, we now specify how abstract states evolve across an ECFG. The intraprocedural analysis computes, for each node, an overapproximation of the possible abstract exception stacks that can arise during execution. This is achieved by defining transfer functions for each kind of node, which propagate abstract states along the edges of the graph. A fixpoint computation over these transfers yields a stable set of abstract states for every node, summarizing all possible behaviors within the function. From these node-level results, we extract the function summary that captures its possible exit behaviors: returns, termination, exceptions thrown and possibly unresolved indirections.

The evolution of abstract states along an ECFG $\langle N, E \cup E_{\mathsf{exc}}, n_0 \rangle$ is described by the *abstract semantics* relation

$$(\to^\sharp) \subseteq (N \times \mathcal{A}) \times \mathcal{A}$$

which relates a node $n \in N$ and the current abstract state $a \in A$ (which only consists of the abstract exception stack) to all new abstract states immediately after executing n.

We assume the *function summary* $\Sigma(f)$ of all callees to be available during the analysis. A function summary is a non-empty set of possible exit behaviors. For the purposes of the abstract interpretation, it suffices to know that the element *terminates* denotes that the called function may terminate the program. Similarly, an element *throws*$*(\tau)$ denotes that an exception of type τ may be thrown. The full definition of function summaries follows in Definition 16.

Definition 12. *(Abstract Semantics) We define the abstract semantics according to the following rules.*

$$\frac{}{(\texttt{instrs}(i_1, \ldots, i_n), \hat{X}) \to^\sharp \hat{X}}$$

$$\frac{\Sigma(f) \neq \{\mathit{terminates}\} \quad \mathit{throws}^*(\tau) \in \Sigma(f)}{(\texttt{call}(f), \hat{X}) \to^\sharp \langle \circlearrowleft, \tau, 0 \rangle \cdot \hat{X}}$$

$$\frac{\epsilon : \tau}{(\texttt{throw}(\epsilon), \hat{X}) \to^\sharp \langle \circlearrowleft, \tau, 0 \rangle \cdot \hat{X}}$$

$$\frac{\hat{X} = \langle \Uparrow, \tau, h \rangle \cdot \hat{X}'}{(\texttt{rethrow}, \hat{X}) \to^\sharp \langle \circlearrowleft, \tau, h \rangle \cdot \hat{X}'}$$

$$\frac{\hat{X} = \langle \nabla, \tau, h \rangle \cdot \hat{X}'}{(\texttt{resume}, \hat{X}) \to^\sharp \langle \circlearrowleft, \tau, h \rangle \cdot \hat{X}'}$$

$$\frac{\tau = \min\{t \in T \mid t_\epsilon \leq t\} \quad \hat{X} = \langle \nabla, \tau_\epsilon, h \rangle \cdot \hat{X}'}{(\texttt{filter}(\tau, T), \hat{X}) \to^\sharp \langle \nabla, \tau_\epsilon, h \rangle \cdot \hat{X}'}$$

$$\frac{\hat{X} = \langle \circlearrowleft, \tau, h \rangle \cdot \hat{X}'}{(\texttt{landing\_pad}, \hat{X}) \to^\sharp \langle \nabla, \tau, h \rangle \cdot \hat{X}'}$$

$$\frac{\hat{X} = \langle \nabla, \tau, h \rangle \cdot \hat{X}'}{(\texttt{begin\_catch}, \hat{X}) \to^\sharp \langle \Uparrow, \tau, h \oplus 1 \rangle \cdot \hat{X}'}$$

$$\frac{\hat{X} = \langle \nabla, \tau, h \rangle \cdot \hat{X}' \quad h \neq 0}{(\texttt{end\_catch}, \hat{X}) \to^\sharp \langle \Uparrow, \tau, h \ominus 1 \rangle \cdot \hat{X}'}$$

$$\frac{\hat{X} = \langle \nabla, \tau, 0 \rangle \cdot \hat{X}'}{(\texttt{end\_catch}, \hat{X}) \to^\sharp \hat{X}'}$$

This definition of abstract semantics is naturally extended to indirect calls where the address of an indirect call is resolved to a set of possible functions, the $\texttt{call}(f)$ case is applied to all possible functions, and their results are joined. If indirection resolution fails, this results in a top state.

Definition 13 (Transfer Function)**.** *We define the* transfer function $\delta : N \times D^\sharp \to D^\sharp$ *by collecting all abstracts states being in semantic relation to each other*

$$\delta(n, d^\sharp) \triangleq \{\hat{a}' \mid (n, \hat{a}) \to^\sharp \hat{a}' \wedge \hat{a} \in d^\sharp\}$$

By iterating δ along the edges of the graph to a fixpoint, we obtain an overapproximation of all abstract states that may arise at each node.

Definition 14 (Abstract Interpretation of an ECFG)**.** *Let $G_e = \langle N, E \cup E_{\mathsf{exc}}, n_0 \rangle$. An abstract interpretation is a mapping*

$$I : N \to D^\sharp$$

assigning to each node a set of abstract states. Let $a_0 = \{[]\}$ be the initial abstract state, representing execution starting with an empty exception stack.

The interpretation I is the least solution to the constraint system

$$I(n_0) \supseteq a_0$$

and for all edges $(n, n') \in E \cup E_{\mathsf{exc}}$,

$$I(n') \supseteq \delta(n, I(n)).$$

The interpretation is computed by initializing all nodes with $\bot$ and iteratively applying the constraints until a fixpoint is reached.

Ensuring Termination. The abstract domain contains infinite ascending chains due to unbounded stack growth and the accumulation of distinct abstract stacks during iteration. To ensure termination, we introduce a widening operator that bounds the height of abstract stacks and forces termination.

Definition 15 (Widening)**.** *Let $D^\sharp_{\mathsf{max}}, \hat{X}_{\mathsf{max}} \in \mathbb{N}$ be fixed bounds. We define a widening operator*

$$\nabla : D^\sharp \times D^\sharp \to D^\sharp$$

as follows. Given two abstract domain elements $a_1, a_2 \in D^\sharp$, first compute the join $a = a_1 \sqcup a_2$. and widen via

$$a_1 \nabla a_2 = \begin{cases} \top & \text{if } |a| > D^\sharp_{max} \vee \exists \hat{X} \in a, |\hat{X}| > \hat{X}_{max} \\ a & \text{else} \end{cases}$$

In practice, bounds $D^\sharp_{\mathsf{max}} = 64$ and $\hat{X}_{\mathsf{max}} = 50$ suffice for the widening to never reach the top state (see Section V).

C. Function Summaries

Definition 16 (Function Summary)**.** *A* function summary *is the non-empty set of possible* exit behaviors *of a function*

$$\Sigma \subseteq \left\{ \begin{matrix} \textit{returns}, & \textit{returns-illegally}, \\ \textit{throws}(\tau), & \textit{throws-illegally}(\tau), \\ \textit{terminates}, & \textit{unresolved} \end{matrix} \right\}$$

We write $\Sigma(f)$ for the function summary of the function with entry node f. For example, $\Sigma(f) = \{\textit{returns}, \textit{throws}(\tau)\}$ means "the function f either returns normally or throws an exception of type τ".

The $*$-*illegally* variants describe behaviors in which the exception stack is left in an invalid state, such as a `return` before the `end_catch` is reached or a `throw` after landing but before the `begin_catch` is reached. We will denote "either *returns* or *returns-illegally*" by *returns** and use the same notation for *throws**. The *unresolved* item is added when an indirect call was not resolved to a sufficiently small set of possible function. It serves as a top state since any behavior could happen, while leaving space for more precise items. For example $\Sigma(f) = \{\textit{returns}, \textit{unresolved}\}$ means "the function f calls an unknown function, but besides that, it returns normally".

These results form the basis for extracting the function summary, which captures the function's possible exit behaviors, including returns, throws, termination, or unresolved indirect calls.

Definition 17 (Exit Nodes)**.** *Given the ECFG $\langle N, E \cup E_{\mathsf{exc}}, n_0 \rangle$ of a function f, we define the set of* exit nodes *$N_{\mathsf{exit}}(f) \subseteq N$ as those nodes where execution exits f. For a node $n \in N$, we have $n \in N_{\mathsf{exit}}(f)$ iff $I(n) \neq \bot$ and any of*

$$n = \texttt{call}(f') \wedge \mathsf{exits}(f') \tag{1}$$
$$n = \texttt{call}(f') \wedge \mathsf{unwinds}(f') \tag{2}$$
$$n = \texttt{return} \tag{3}$$
$$n = \texttt{terminate} \tag{4}$$
$$n = \texttt{throw}(\epsilon) \wedge \mathsf{noLP}(n, n') \tag{5}$$
$$n = \texttt{rethrow} \wedge \mathsf{noLP}(n, n') \tag{6}$$
$$n = \texttt{resume} \wedge \mathsf{noLP}(n, n') \tag{7}$$

where

$$\mathsf{exits}(f) \Leftrightarrow \{\textit{terminates}, \textit{unresolved}\} \cap \Sigma(f) \neq \emptyset$$
$$\mathsf{unwinds}(f) \Leftrightarrow \textit{throws}^*(\tau) \in \Sigma(f) \wedge \mathsf{noLP}(n, n')$$
$$\mathsf{noLP}(n, n') \Leftrightarrow \nexists n'. \langle n, n' \rangle \in N_{\mathsf{exc}}$$

Here, cases 2,5,6, and 7 correspond to exception unwinding without a landing pad in the current function.

Definition 18 (Function Summary Computation)**.** *We compute the function summary $\Sigma(f)$ by summarizing all of f's exit nodes*

$$\Sigma(f) \triangleq \bigcup_{n \in N_{\mathsf{exit}}(f)} \Sigma_N(n)$$

where function $\Sigma_N : N \to \Sigma$ summarizes a node n. Figure 3 shows the defining equations for each type of exit node.

Handling Recursion. When analyzing (mutually) recursive functions, summaries may be required before they have been computed. In particular, if a function f is encountered a second time during the summary computation of one of its (direct or indirect) callers, we cannot perform the full abstract interpretation again without creating a recursive dependency. In this case, we instead construct a syntactic over-approximation of $\Sigma(f)$: we scan all instructions of f, ignoring reachability, and collect any behavior that the function may exhibit purely based on its syntax. Specifically, encountering a `return` instruction adds *returns*, encountering a terminating instruction adds *terminates*, and encountering a `throw`(ϵ) adds the corresponding *throws*(τ). This initial approximation breaks the recursive cycle and allows the caller's analysis to proceed. Afterwards, once the caller's summary has been computed, we re-analyze f using the standard abstract interpretation to obtain a precise summary.

D. Correctness

To ensure that our abstract interpretation is sound with respect to the concrete semantics of exceptional execution, we relate concrete and abstract states via a standard abstraction framework. This allows us to reason about exceptional program behavior at the level of abstract states while preserving correctness with respect to all possible concrete executions.

$$
\begin{array}{lll}
\Sigma_N(\texttt{call}(f')) & = \{\mathit{terminates}\} & \text{if } \mathit{terminates} \in \Sigma(f') \\
& \cup \{\mathit{unresolved}\} & \text{if } \mathit{unresolved} \in \Sigma(f') \\
& \cup \{\mathit{throws}(\tau)\} & \text{for each } \mathit{throws}^*(\tau) \in \Sigma(f') \\
\Sigma_N(\texttt{return}) & = \{\mathit{returns}\} & \text{if } [] \in I(n) \\
& \cup \{\mathit{returns\text{-}illegally}\} & \text{if } \exists \hat{X} \in I(n).\hat{X} \neq [] \\
\Sigma_N(\texttt{terminate}) & = \{\mathit{terminates}\} & \\
\Sigma_N(\texttt{throw}(\epsilon)) & = \{\mathit{throws}(\tau)\} & \text{if } \epsilon : \tau \text{ and } \exists \hat{X} \in I(n).\hat{X} \neq \langle \triangledown, \tau', h\rangle \cdot \hat{X}' \\
& \cup \{\mathit{throws\text{-}illegally}(\tau)\} & \text{if } \epsilon : \tau \text{ and } \exists \hat{X} \in I(n).\hat{X} = \langle \triangledown, \tau', h\rangle \cdot \hat{X}' \\
\Sigma_N(\texttt{rethrow}) & = \{\mathit{throws}(\tau)\} & \text{if } \exists \hat{X} \in I(n).\hat{X} = \langle \Uparrow, \tau', h\rangle \cdot \hat{X}' \\
\Sigma_N(\texttt{resume}) & = \{\mathit{throws}(\tau)\} & \text{if } \exists \hat{X} \in I(n).\hat{X} = \langle \triangledown, \tau', h\rangle \cdot \hat{X}'
\end{array}
$$

Fig. 3: Function summary computation based on the exit node type.

We define an abstraction function $\alpha : D \to D^\sharp$ and a concretization function $\gamma : D^\sharp \to D$ forming the bridge between concrete execution states and their abstract representation. The definitions for both functions are straightforward and directly induced by the structure of the abstract and concrete states. Appendix A defines them and shows the correctness theorems in more detail.

Theorem 1 (Galois Connection)**.** *The abstraction and concretization functions form a Galois connection between $(D, \sqsubseteq_C)$ and $(D^\sharp, \sqsubseteq_A)$, i.e., for all $d \in D$ and all $a^\sharp \in D^\sharp$:*

$$\alpha(d) \sqsubseteq_A a^\sharp \;\Leftrightarrow\; d \sqsubseteq_C \gamma(a^\sharp)$$

Theorem 2 (Local Soundness)**.** *All intraprocedural transitions of the concrete semantics are soundly over-approximated by the abstract transfer function δ, i.e., function calls are excluded from this theorem.*

Let $G_e = \langle N, E \cup E_{\text{exc}}, n_0\rangle$ be the ECFG for a function f, and let $n, n' \in N \wedge n \notin N_{exit}(f)$, where n is not of the form $\texttt{call}(f')$*, and $c, c' \in \mathcal{C}$. Then:*

$$(n, c) \to (n', c') \;\Rightarrow\; c' \in \gamma(\delta(n, \alpha(\{c\}))).$$

Soundness intuition for function summaries. Function summaries $\Sigma(f)$ depend on the correctness of the intraprocedural abstract semantics δ established in Theorem 2. Summaries capture the behavior of a function at its exit nodes and the propagation of exceptions beyond handler boundaries by classifying states according to their exception stack. By construction, $\Sigma(f)$ over-approximates all such exit-node behaviors and unhandled exception flows induced by δ. Consequently, call sites are soundly interpreted via $\Sigma(f)$, since all behaviors of the callee that may affect the caller are included in its summary.

V. Evaluation

We evaluate our abstract interpretation framework for exceptional control flow along two main dimensions: (1) applicability and scalability to real-world binaries, and (2) correctness of the inferred exception behavior. For the first, we report on runtimes relative to binary sizes, as well as the sizes of the generated ECFGs. For the second, we report and discuss the models inferred by our approach. A challenge here is that typically a ground truth is unavailable.

Our evaluation is conducted on a corpus of 386 real-world binaries, consisting of shared libraries (`.so`) and standalone executables. In total, the corpus comprises roughly 18M machine instructions in 170K functions, providing a diverse and realistic workload for assessing both scalability and behavioral coverage. All experiments are performed on a MacBook Pro M1. The implementation of our abstract interpretation framework is realized as an extension of FoxDec [15], a disassembler that recovers a control-flow graph from a binary. FoxDec's recovered CFG serves as the input to our extension, which reads the binary's exception-handling metadata to construct the corresponding ECFG. The abstract interpretation is then run on the resulting ECFG.

The Boost C++ Libraries are a large collection of extensions to the C++ Standard Library. ROS2 is a Robot Operating System. Geant4 is a toolkit developed by CERN for simulation of particle behavior. MySQL is a well-known RDBMS. We have applied our approach to a set of miscellaneous binaries, including a ws-proxy server and programs for image analysis, as well as a set of microbenchmarks containing manually constructed intricate exceptional control flow.

A. Applicability and Scalability.

Runtime performance. Lifting, generating ECFGs, and running abstract interpretation to derive function summaries takes on the order of seconds to minutes per binary (see Table I). For example, all of Geant4 (7.1M instructions) takes roughly 40 minutes.

ECFG sizes. The ECFGs proposed in this paper are relatively compact representations of exceptional control flow. Overall, ECFGs are about 5.2× smaller than normal CFGs. Figure 4 shows that many functions reduce to an ECFG of size two or three (i.e., only an entry- and one or two exit-points), and CFGs with hundreds of nodes may be compressed to only a few. In some cases, however, an ECFG becomes slightly larger: all blocks are relevant to

Project	# Binaries	# Instrs	# Funcs	Runtime (mm:ss)	Source
Boost	*43*	*1.1M*	*10.8K*	*02:42*	https://github.com/boostorg/boost
ROS2	*243*	*3.5M*	*53.2K*	*20:05*	https://github.com/ros2/ros2
Geant4	*33*	*7.1M*	*6.1K*	*39:30*	https://geant4.web.cern.ch
MySQL	*30*	*0.5M*	*45.7K*	*01:31*	https://github.com/mysql/mysql-server
Misc.	*19*	*5.7M*	*54.4K*	*11:39*	—
Microbenchmarks	*18*	*6.9K*	*264*	*00:04*	—
Total	386	18M	170K	1:15:31	—

TABLE I: Overview of the evaluated binary corpus.

the exceptional control flow, and some basic blocks needed to be split since they contained relevant function calls.

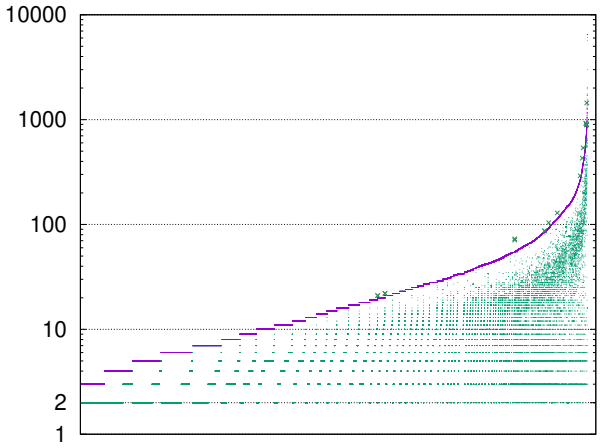

Fig. 4: Comparison of original CFG sizes (purple) versus ECFG sizes (green) across all evaluated binaries. Both lines consists of 170K points. The green crosses indicate cases where the ECFG is larger than the CFG.

Overall, the results show that our analysis is applicable to real-world binaries and can be executed efficiently on large codebases. The combination of function-level abstraction and a compressed control-flow representation enables scalable analysis even for large software artifacts.

B. Correctness of the inferred exception behavior.

Unlike traditional program analysis settings, there is no reliable ground truth available for exception behavior in stripped binaries. Source code is often unavailable or does not precisely reflect compiled behavior, and `noexcept` annotations are not reliable indicators of whether exceptions may be thrown in the binary. Similarly, documentation is incomplete or outdated for many libraries in our corpus.

We therefore evaluate correctness through (i) large-scale classification of observed function behaviors, and (ii) targeted manual inspection of representative cases exhibiting interesting or unexpected behavior.

Summary of inferred function behavior. Roughly two thirds of all functions return normally, while about one third may either return or throw depending on execution

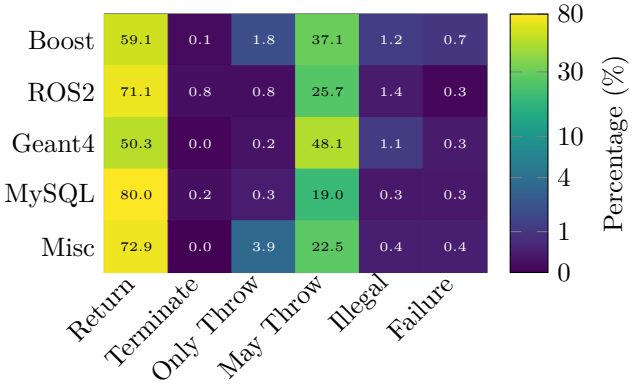

Fig. 5: Function-level exception behavior distribution across projects. Values represent percentage of functions per category.

context. The remaining categories (always throwing, always terminating, illegal behavior, and analysis failures) together account for a small fraction of the total (Figure 5). This distribution is consistent across all non-microbenchmark projects, indicating that the observed pattern reflects real-world C++ codebases rather than project-specific artifacts.

Illegal exception behavior. A small fraction of functions exhibit "illegal" exception behavior, capturing control-flow patterns that violate the standard structure of C++ exception handling as modeled in our abstract semantics. Such cases arise in two situations: when a return instruction is reachable before the active catch region has been fully exited, or when a new exception is raised during stack unwinding before reaching a landing pad. The latter is typically caused by destructors that may throw during unwinding, which the C++ model defines as triggering program termination.[8] Manual inspection suggests these cases arise when the program is already in a failure state where recovery is not expected, rather than on normal exception-handling paths.

Manual inspection. We manually inspected a representative subset of functions exhibiting exceptional or illegal behavior, tracing exception propagation through the disassembled control flow to verify that inferred classifications align with observable execution paths. We additionally

[8] https://en.cppreference.com/cpp/language/throw

observed that some functions annotated `noexcept` at the source level still contain reachable exception-throwing behavior in the binary, confirming that such annotations do not reliably reflect compiled control-flow semantics.

Limitations of CFG reconstruction. A small number of analysis failures are caused by incomplete CFG reconstruction in the underlying FoxDec disassembler, preventing abstract interpretation from being executed. Similarly, a subset of illegal classifications are false positives arising from conservative control-flow reconstruction, where infeasible paths in the reconstructed CFG lead to spurious exception propagation scenarios.

Summary. The inferred exception behaviors are consistent with observable control-flow structure across all evaluated projects. The majority of functions exhibit stable return behavior, while a substantial fraction includes conditional exception propagation. Rare illegal cases correspond to well-understood edge cases such as unwinding constraint violations. The combination of large-scale statistical analysis and manual inspection provides strong evidence that the inferred behaviors are meaningful at the binary level.

VI. Conclusion

Exception handling introduces control-flow behavior that is often ignored by binary analyses despite being an essential component of modern compiled software. In this work, we presented an abstract interpretation for exception-aware control-flow graphs and used it to compute exception-aware function summaries directly from binaries.

Our approach extends existing binary control-flow reconstruction with explicit representations of exceptional control flow and provides a formal framework for reasoning about exception behavior at the binary level. We proved the correctness of the analysis, implemented it in FoxDec, and evaluated it on real-world binaries. The results show that exception-aware summaries can be computed efficiently in practice and that the resulting summaries capture a range of behaviors, including normal exception propagation, termination, and exceptional situations that violate language-level expectations.

Overall, our results demonstrate that exception handling can be treated as a first-class component of binary abstract interpretation, enabling more precise reasoning about software that relies on C++ exceptions.

Acknowledgements

This work is supported by the Defense Advanced Research Projects Agency (DARPA) under Prime Contract No. HR001124C0492 and US National Science Foundation (NSF) under CNS-2234257. Any views, opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of DARPA, NSF, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints of publications for Government purposes notwithstanding any copyright notation hereon.

References

[1] Kapil Anand et al. “A compiler-level intermediate representation based binary analysis and rewriting system”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 295–308. ISBN: 9781450319942. DOI: 10.1145/2465351.2465380.

[2] Joshua Bockenek, Freek Verbeek, and Binoy Ravindran. “Exceptional Interprocedural Control Flow Graphs for x86-64 Binaries”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 21st International Conference, DIMVA 2024, Lausanne, Switzerland, July 17–19, 2024, Proceedings*. Lausanne, Switzerland: Springer-Verlag, 2024, pp. 3–22. ISBN: 978-3-031-64170-1. DOI: 10.1007/978-3-031-64171-8_1.

[3] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973.

[4] Alexander Fokin et al. “SmartDec: Approaching C++ Decompilation”. In: *2011 18th Working Conference on Reverse Engineering*. 2011, pp. 347–356. DOI: 10.1109/WCRE.2011.49.

[5] Michael Gschwind and Erik R. Altman. “Precise Exception Semantics in Dynamic Compilation”. In: *Proceedings of the 11th International Conference on Compiler Construction*. CC ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 95–110. ISBN: 3540433694. DOI: 10.1007/3-540-45937-5_9.

[6] Graham Hutton and Joel Wright. “Compiling Exceptions Correctly”. In: *Mathematics of Program Construction*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 211–227. ISBN: 978-3-540-27764-4. DOI: 10.1007/978-3-540-27764-4_12.

[7] Maria Kechagia et al. “Effective and efficient API misuse detection via exception propagation and search-based testing”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 192–203. ISBN: 9781450362245. DOI: 10.1145/3293882.3330552.

[8] Johannes Kinder and Dmitry Kravchenko. "Alternating Control Flow Reconstruction". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Viktor Kuncak and Andrey Rybalchenko. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 267–282. ISBN: 978-3-642-27940-9. DOI: 10.1007/978-3-642-27940-9_18.

[9] Johannes Kinder, Florian Zuleger, and Helmut Veith. "An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries". In: Jan. 2009, pp. 214–228. ISBN: 978-3-540-93899-6. DOI: 10.1007/978-3-540-93900-9_19.

[10] Prakash Prabhu et al. "Interprocedural Exception Analysis for C++". In: *ECOOP 2011 – Object-Oriented Programming*. Ed. by Mira Mezini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 583–608. ISBN: 978-3-642-22655-7. DOI: 10.1007/978-3-642-22655-7_27.

[11] P. Ročkai, J. Barnat, and L. Brim. "Model checking C++ programs with exceptions". In: *Science of Computer Programming* 128 (2016). Special issue on Automated Verification of Critical Systems (AVoCS'14), pp. 68–85. ISSN: 0167-6423. DOI: 10.1016/j.scico.2016.05.007.

[12] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York, NY: New York Univ. Comput. Sci. Dept., 1978. URL: https://cds.cern.ch/record/120118.

[13] Yan Shoshitaishvili et al. "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[14] Vladimír Štill, Petr Ročkai, and Jiří Barnat. "Using Off-the-Shelf Exception Support Components in C++ Verification". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 54–64. DOI: 10.1109/QRS.2017.15.

[15] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. "Sound C Code Decompilation for a Subset of x86-64 Binaries". In: *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings*. Amsterdam, The Netherlands: Springer-Verlag, 2020, pp. 247–264. ISBN: 978-3-030-58767-3. DOI: 10.1007/978-3-030-58768-0_14.

[16] Hao Zhang et al. "Detecting Exception Handling Bugs in C++ Programs". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 1084–1095. DOI: 10.1109/ICSE48619.2023.00098.

Appendix

Correctness Of Abstract Interpretation

This appendix gives explicit definitions of the abstraction and concretization functions α and γ, and proof sketches for the Galois connection and the local soundness of the abstract transfer function δ.

Definition 19 (Abstraction)**.** *We define the* abstraction $\alpha : D \to D^\sharp$ *in terms of abstracting the constituents via the state abstraction* $\alpha_C : \mathcal{C} \to \mathcal{A}$*, the exception stack abstraction* $\alpha_X : ExStack \to \widehat{ExStack}$*, the exception item abstraction* $\alpha_e : Ex \to \widehat{Ex}$*, and the handler count abstraction* $\alpha_h : \mathbb{N} \to \hat{H}$*.*

$$
\begin{array}{lcl}
\alpha(\{c_1, \ldots, c_n\}) & \triangleq & \{\alpha_C(c_1), \ldots, \alpha_C(c_n)\} \\
\alpha_C(\langle \mu, \Gamma, X \rangle) & \triangleq & \alpha_X(X) \\
\alpha_X([e_1, \ldots, e_n]) & \triangleq & [\alpha_e(e_1), \ldots, \alpha_e(e_n)] \\
\alpha_e(\langle m, \epsilon, h \rangle) & \triangleq & \langle m, \tau, \alpha_h(h) \rangle \\
 & & \quad \textit{where } \epsilon : \tau \\
\alpha_h(h) & \triangleq & \begin{cases} h & \textit{if } h \leq \hat{h}_{\mathsf{max}} \\ \top & \textit{else} \end{cases}
\end{array}
$$

Definition 20 (Concretization)**.** *The* concretization $\gamma : D^\sharp \to D$ *is defined symmetrically via the state concretization* $\gamma_A : \mathcal{A} \to \mathcal{P}(\mathcal{C})$*, the exception stack concretization* $\gamma_X : \widehat{ExStack} \to \mathcal{P}(ExStack)$*, the exception item concretization* $\gamma_e : \widehat{Ex} \to \mathcal{P}(Ex)$*, and the handler count concretization* $\gamma_h : \hat{H} \to \mathcal{P}(\mathbb{N})$*.*

$$
\begin{array}{lcl}
\gamma(\{a_1, \ldots, a_n\}) & \triangleq & \bigcup\{\gamma_A(a_1), \ldots, \gamma_A(a_n)\} \\
\gamma_A(\hat{X}) & \triangleq & \{\langle \mu, \Gamma, X \rangle \mid X \in \gamma_X(\hat{X})\} \\
 & & \quad \textit{for all } \mu, \Gamma \\
\gamma_X([\hat{e}_1, \ldots, \hat{e}_n]) & \triangleq & \{[e_1, \ldots, e_n] \mid e_i \in \gamma_e(\hat{e}_i)\}\} \\
\gamma_e(\langle m, \tau, \hat{h} \rangle) & \triangleq & \{\langle m, \epsilon, h \rangle \mid h \in \gamma_h(\hat{h})\} \\
 & & \quad \textit{for all } \epsilon : \tau \\
\gamma_h(\hat{h}) & \triangleq & \begin{cases} \{\hat{h}\} & \textit{if } \hat{h} \neq \top \\ \mathbb{N} & \textit{else} \end{cases}
\end{array}
$$

Theorem 1 (Galois Connection)**.** *The abstraction and concretization functions form a Galois connection between* $(D, \sqsubseteq_C)$ *and* $(D^\sharp, \sqsubseteq_A)$*, i.e., for all* $d \in D$ *and all* $a^\sharp \in D^\sharp$*:*

$$\alpha(d) \sqsubseteq_A a^\sharp \;\Leftrightarrow\; d \sqsubseteq_C \gamma(a^\sharp).$$

Proof. The abstraction and concretization functions are defined component-wise over the structure of states, stacks, and exceptions. We therefore prove the statement by structural decomposition over the constituents.

We first observe that each pair of component functions (α_C, γ_A), (α_X, γ_X), (α_e, γ_e), and (α_h, γ_h) forms a Galois connection on its respective domain. This follows directly from their definitions, which either map elements to their abstract representatives or to the set of corresponding concrete values.

Since D and $D^\sharp$ are built as product-like structures over these components, the abstraction α and concretization γ act pointwise on each component. The ordering $\sqsubseteq_C$ and $\sqsubseteq_A$ is also defined component-wise (with set inclusion on collections and pointwise lifting on structured elements).

Thus, for any $d \in D$ and $a^\sharp \in D^\sharp$, the relation

$$\alpha(d) \sqsubseteq_A a^\sharp$$

holds if and only if each component of $\alpha(d)$ is below the corresponding component of $a^\sharp$, which is equivalent to each concrete component of d being contained in the concretization of the corresponding abstract component of $a^\sharp$. This is exactly

$$d \sqsubseteq_C \gamma(a^\sharp).$$

Hence, the abstraction and concretization functions form a Galois connection. □

Theorem 2 (Local Soundness)**.** *All intraprocedural transitions of the concrete semantics are soundly over-approximated by the abstract transfer function* δ*, i.e., function calls are excluded from this theorem.*

Let $G_e = \langle N, E \cup E_{\mathsf{exc}}, n_0 \rangle$ *be the ECFG for a function* f*, and let* $n, n' \in N \wedge n \notin N_{exit}(f)$*, where* n *is not of the form* `call`(f')*, and* $c, c' \in \mathcal{C}$*. Then:*

$$(n, c) \to (n', c') \;\Rightarrow\; c' \in \gamma(\delta(n, \alpha(\{c\}))).$$

Proof. We proceed by case analysis on the concrete transition $(n, c) \to (n', c')$. Since n is not a call node, only intraprocedural rules of the concrete semantics (Figure 2) are applicable.

Case Instr. The concrete rule requires a normal (non-unwinding) configuration, so only instructions that leave the call and exception stacks unchanged are executed; δ preserves this structure and is a no-op on the exception stack. Soundness then follows from Theorem 1.

Case Throw, Rethrow, Resume. These rules manipulate the exception stack and propagate exceptional control flow; the abstract semantics mirrors these updates. By Theorem 1, the resulting state is covered by $\gamma(\delta(\cdot))$.

Case Landing, Filter, Begin-Catch, End-Catch. These rules describe structured exception handling; each concrete transition has a corresponding rule in δ that over-approximates landing-pad selection, filtering, and handler execution, with soundness following from the component-wise Galois connection.

Case Call, Return, Unwind. These cases model inter-procedural control transfer and stack unwinding and are thus excluded from this theorem. □