QUERYX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries

HyungSeok Han[†*], JeongOh Kyea[†], Yonghwi Jin[†], Jinoh Kang[†], Brian Pak[†], Insu Yun[*]
[†]Theori Inc., [*]KAIST

***Abstract*—Extensible static checking tools, such as Sys and CodeQL, have successfully discovered bugs in source code. These tools allow analysts to write application-specific rules, referred to as *queries*. These queries can leverage the domain knowledge of analysts, thereby making the analysis more accurate and scalable. However, the majority of these tools are inapplicable to binary-only analysis. One exception, joern, translates a binary code into decompiled code and feeds the decompiled code into an ordinary C code analyzer. However, this approach is not sufficiently precise for symbolic analysis, as it overlooks the unique characteristics of decompiled code. While binary analysis platforms, such as angr, support symbolic analysis, analysts must understand their intermediate representations (IRs) although they are mostly working with decompiled code.**

In this paper, we propose a precise and scalable symbolic analysis called fearless symbolic analysis that uses intuitive queries for binary code and implement this in QUERYX. To make the query intuitive, QUERYX enables analysts to write queries on top of decompiled code instead of IRs. In particular, QUERYX supports callbacks on decompiled code, using which analysts can control symbolic analysis to discover bugs in the code. For precise analysis, we lift decompiled code into our IR named DNR and perform symbolic analysis on DNR while considering the characteristics of the decompiled code. Notably, DNR is only used internally such that it allows analysts to write queries regardless of using DNR. For scalability, QUERYX automatically reduces control-flow graphs using callbacks and ordering dependencies between callbacks that are specified in the queries. We applied QUERYX to the Windows kernel, the Windows system service, and an automotive binary. As a result, we found 15 unique bugs including 10 CVEs and earned $180,000 from the Microsoft bug bounty program.

1. Introduction

Static analysis is one of the most widely used techniques for finding bugs in code [2, 5, 6, 16, 21, 38–40, 51, 59, 60, 66–69, 72, 77–81]. Unlike dynamic analysis (e.g., fuzzing), static analysis can achieve high code coverage without the difficulty of obtaining inputs for high code coverage. Despite its advantages, static analysis fundamentally suffers from the trade-off between scalability and accuracy. More specifically, static analysis requires to be path- and context-sensitive to be accurate; however, this introduces serious scalability issues, including path explosion. If we employ approximations to address these issues, static analysis generates many false alarms due to over- or under-approximations.

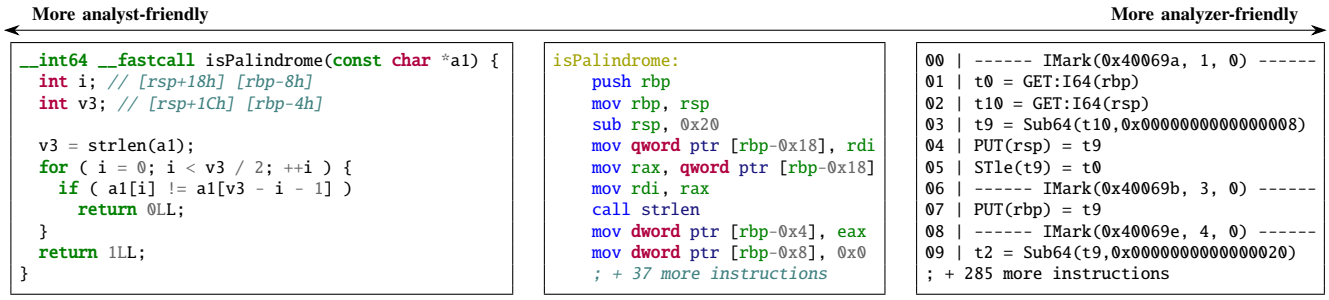
	Input	Analysis objects	Syntactic	Data-flow	Symbolic
CodeQL [22]	Source	Source	✓	✓	×
Sys [7]	Source	LLVM IR	✓	✓	✓
joern [76]	Source Binary	Source Decompiled code	✓	✓	×
angr [56]	Binary	VEX IR	✓	✓	✓
BAP [9]	Binary	BIL	✓	✓	✓
QUERYX	Binary	Decompiled code	✓	✓	✓

TABLE 1: Comparison between extensible static checking tools. Notably, QUERYX is the first tool that supports symbolic analysis on binary code using decompiled code.

To address this trade-off, many extensible static checking tools (e.g., CodeQL [22], Sys [7], and joern [76]) have been proposed. The key idea of these approaches is to utilize the domain knowledge of analysts about the target application. In particular, analysts can conduct static analysis with precise and application-specific rules on top of these tools, rather than relying on generic approximations alone. Furthermore, these tools often provide a convenient method of writing application-specific rules, which are called *queries*. They also enable application-aware static analysis, such as syntactic matching, data-flow analysis, and symbolic analysis based on a given query, as shown in Table 1.

While these tools have achieved considerable success in source code analysis, they are inefficient in analyzing binaries. Except for joern, most extensible static checking tools, such as Sys and CodeQL, can only be successfully applied to source code. In contrast, joern supports binary analysis on decompiled code using a fuzzy C parser; however, this tool lacks support for symbolic analysis, which is required for complex analyses. Moreover, these tools often require complicated rules to be scalable analysis. For instance, Sys requires more than 300 lines of Haskell code for heap out-of-bound analysis. In addition to these source code analysis tools, there exist binary symbolic analysis tools such as angr [55] and BAP [9]. However, they have different design philosophies that aim at supporting low-level features and extensibility for developing other tools. They rely on an intermediate representation (IR) to manage complexity and support multiple architectures. Unfortunately, this requires analysts to understand the IR generated from binary code, even if analysts are mostly working with decompiled code.

In this paper, we propose a precise and scalable symbolic analysis with intuitive queries for binaries, which is called *fearless symbolic analysis*. We implement this in QUERYX, which allows analysts to write queries on decompiled code. This is motivated by that decompiled code is friendlier to binary analysts than IR. And QUERYX supports callbacks



(a) Decompiled code. **(b)** Assembly code. **(c)** Intermediate representation.

Figure 1: Different representations of a binary code that checks palindrome: decompiled code, assembly code, and intermediate representation (IR) from angr [56]. Decompiled code is more analyst-friendly than the others whereas IR is the most analyzer-friendly.

on abstract syntax tree (AST) nodes of the decompiled code. These callbacks will be invoked after the corresponding AST nodes are evaluated during symbolic execution. Using these callbacks, analysts can capture or validate the desired properties. Notably, QUERYX supports JavaScript-like queries, which are expected to reduce the learning curve of QUERYX.

Although the high-level idea is intuitive, achieving the fearless symbolic analysis is nontrivial. For precise analysis, QUERYX considers the unique aspects of the decompiled code generated from a binary code. In particular, decompiled code differs from an ordinary C program in the presence of binary-dependent code (e.g., offset-based memory access), binary-embedded data (e.g., global variables), and decompiler-induced code (e.g., LOBYTE macro in IDA Hex-Rays [28]). To address this difference, we devise DNR, a new IR lifted from decompiled code while preserving the semantics of the binary code. By considering the semantics of binary code in DNR, QUERYX can perform binary-aware and precise symbolic analysis. In addition, DNR includes which AST node is lifted to it. Therefore, this allows analysts to write queries based on decompiled code instead of DNR.

For scalable symbolic analysis, QUERYX performs under-constrained symbolic execution, which starts the analysis from function entries as UC-KLEE [52]; however, this is insufficient for large binaries. QUERYX thus utilizes its callback-based analysis where callbacks ensure flexible analysis and inherently represent interesting nodes for analysis. Thus, QUERYX can filter out uninteresting paths that do not include callbacks. In addition, QUERYX supports ordering dependencies between callbacks. These dependencies enable QUERYX to prune paths automatically without any complicated queries. This imparts scalability to QUERYX for supporting real-world binaries, such as the Windows kernel.

We applied QUERYX to COTS binaries: the Windows kernel, the Windows system service, and an automotive binary. And we wrote four queries for four types of bugs: heap overflow, kernel memory address disclosure, path traversal, and out-of-bound access. As a result, we found 15 previously unknown bugs including 10 CVEs, and earned $180,000 from the Microsoft bug bounty program. We also present that QUERYX beat existing extensible static checking tools in terms of query writing and bug findings. And we experimentally show the effectiveness of our path pruning.

In summary, our main contributions are as follows:

- We propose fearless symbolic analysis, a precise and scalable symbolic analysis with intuitive queries for binaries based on decompiled code, and implemented it in QUERYX.
- We present binary-aware analysis to precisely analyze decompiled code while preserving the semantics of the unique aspects of decompiled code.
- We effectively improve the scalability of symbolic analysis with intuitive queries that specify callbacks on AST nodes of decompiled code and ordering dependencies between them.
- By evaluating QUERYX on the Windows kernel, the Windows system service, and an automotive binary, we found 15 previously unknown bugs including 10 CVEs, and earned $180,000 from Microsoft as a bug bounty.

2. Goals and Approaches

This section discusses the goals and approaches for accomplishing the proposed fearless symbolic analysis.

2.1. Analyst-friendly Query

As previously mentioned, static analysis is powerful but inherently unscalable. To overcome this issue, extensible static checking tools rely on human capabilities. Analysts manually create queries that guide static analysis to avoid uninteresting paths and suppress false alarms. However, this approach is not straightforward when handling binaries because of inconsistent representations by analysts and tools. Unlike source-based analysis tools, binary-based analysis tools often use IR. IR avoids the complexity of machine code (e.g., several instruction types and side effects), simplifies the analysis, and provides multi-architecture support. Unfortunately, IR is not analyst-friendly (see, Figure 1), and analysts always prefer to read decompiled code, which contains more analyst-friendly information, such as types and high-level control flows. Because of this inconsistency, when analysts write IR-based queries, they must encode their understandings in decompiled code into IR, thereby complicating their query writing.

```
1  int i; // [rsp+18h] [rbp-58h]
2  int v1; // [rsp+1Ch] [rbp-54h]
3  int s[3]; // [rsp+20h] [rbp-50h]
4  int v2; // [rsp+2Ch] [rbp-44h]
5
6  memset(s, 0, 0x40LL);
7
8  // ISSUE1: Binary-embedded data
9  v2 = dword_2010A4;
10 for ( i = 0; i <= 15; ++i ) {
11   // ISSUE2: Binary-dependent code
12   //   (e.g., out-of-bounds due to failed type inferences)
13   // ISSUE3: Decompiler-induced code (e.g., LOBYTE)
14   v1 += LOBYTE(s[i]);
15 }
```

Figure 2: Examples that illustrate the issues in symbolic analysis on decompiled code. Unlike ordinary C programs. decompiled code requires extra attention due to the presence of binary-embedded data, binary-dependent code, and decompiler-induced code.

Our Approach: Query on Decompiled Code. To address this issue, QUERYX allows analysts to write queries directly based on decompiled code. Furthermore, QUERYX supports analysts to register callbacks on AST nodes of decompiled code using the corresponding high-level information, such as variable names and type information, to check desired properties while following the program execution. Notably, callbacks are triggered right after the symbolic executor evaluates the corresponding AST nodes.

Using decompiled code has several advantages. First, it is architecture-agnostic, similar to IR, which naturally supports multiple architectures. Therefore, we successfully applied QUERYX to the x86-based Windows kernel and an ARM-based automotive binary. Second, decompiled code is more human-readable, which improves usability, and allows interactive analysis. If analysts modify the decompiled code, QUERYX can automatically leverage this change in its analysis. Assume that QUERYX reports a false alarm for buffer overflows because a decompiler incorrectly infers the size of a certain variable. In this case, analysts can suppress this false alarm by correcting the variable's type information in the decompiled code instead of adding a special routine to a query. Moreover, if analysts assign a name to a certain binary location for improved understanding, they can reuse that name to write more analyst-friendly queries in QUERYX.

2.2. Precise Analysis on Decompiled Code

The decompiled code is similar to ordinary C code. Therefore, joern [76] treats decompiled code identically to standard C code. To analyze a binary code, joern first decompiles the binary code using Ghidra [50] or IDA Hex-Rays and then applies the existing analysis for C code. It is enough for syntactic analysis but not for semantic or symbolic analysis due to the characteristics of the decompiled code.

Decompiled code has characteristics that differentiate it from standard C, and these characteristics should be carefully managed for precise symbolic analysis. Figure 2 highlights several issues in the symbolic analysis on decompiled code using examples. First, the decompiled code uses binary-embedded data (Line 9), particularly for global variables. Symbolic analysis may result in incorrect findings without properly modeling the environment for a binary. Second, the decompiled code includes binary-dependent code, which implicitly assumes memory locations. For example, in Line 14, this program seems to have out-of-bound accesses for `s`, the length of which is three (Line 3). According to C specifications, this is an undefined behavior that can result in any consequence. However, this is erroneous reasoning caused by incorrect type inferences of the decompiler; the length of `s` is `0x10`, as shown by the `memset` call (Line 6). Therefore, for precise analysis, our system should be aware of such implicit assumptions regarding memory locations. Finally, decompilers extensively induce special functions without explicitly including them. For example, IDA Hex-Rays uses `LOBYTE`, which is a Windows macro [44] that returns the last byte of a given argument, even in the Linux code. This enables decompilers to readily translate binary code into valid C code; however, QUERYX must model them appropriately for symbolic analysis.

Our Approach: Binary-aware Analysis. To address this, QUERYX conducts a binary-aware symbolic analysis on the decompiled code. For this, QUERYX first lifts the decompiled code into our IR, named DNR, and performs symbolic analysis on it. DNR simplifies the analysis of QUERYX by maintaining binary and decompiled code information (e.g., the corresponding AST node). Unlike other IRs, DNR is strongly connected to the decompiled code. Therefore, analysts can work with decompiled code via the AST without directly handling DNR. Subsequently, when QUERYX performs memory operations on global variables, it fetches binary-embedded data from DNR to accurately model them. In addition, DNR includes the memory addresses of the variables in the decompiled code. This allows QUERYX to correctly represent binary-dependent code that seemingly violates C specifications. Moreover, QUERYX lifts decompiler-induced routines (e.g., `LOBYTE`) to semantically equivalent DNR. This eliminates the need for QUERYX to consider such routines in subsequent symbolic analysis. Thus, QUERYX can considerably increase the precision of the symbolic analysis on the decompiled code.

2.3. Scalable Analysis with Intuitive Query

Symbolic analysis suffers from scalability issues due to path explosions. To improve the scalability, Sys [7] suggested a two-step analysis combined with fine-grained static analysis. Sys first performs static analysis to select only interesting paths and verifies them by symbolic analysis. This idea is intuitive and effective. However, this burdens analysts by requiring them to write two non-trivial queries: one for static analysis and the other for symbolic analysis. These queries should be written for LLVM IR, which is less intuitive than the source code or decompiled code. For example, Sys uses 273 lines for static analysis and 62 lines for symbolic analysis to detect heap out-of-bound bugs. Occasionally, this complexity can cause Sys to miss critical bugs. For instance, we discovered that Sys missed a critical heap out-of-bound bug due to its incomplete static analysis query (see §8.2).

Our Approach: CFG Reduction by Callbacks and their Dependencies. QUERYX automatically scales its symbolic analysis using callbacks and their dependencies, which analysts write on the queries. As previously mentioned, QUERYX supports registering callbacks on AST nodes for the symbolic analysis. These callbacks inherently indicate which nodes are important for the analysis. In addition, QUERYX allows analysts to specify the ordering dependencies between callbacks (e.g., `malloc` → `memcpy`), which are common in bug findings. Using this information, QUERYX reduces the control-flow graph (CFG) by eliminating the nodes disconnected from AST nodes that have callbacks and are unable to satisfy the specified ordering dependencies.

3. Overview

The primary goal of QUERYX is to symbolically analyze decompiled code based on queries for finding bugs in COTS binaries. In this section, we demonstrate how QUERYX finds bugs with a running example and present the overall architecture of QUERYX.

3.1. Running Example

To demonstrate the overall procedure of QUERYX, we describe how QUERYX discovered a new heap overflow bug in the Windows kernel (CVE-2021-41378) from an existing bug (CVE-2021-31979 [47]), which was actively exploited in the wild according to Microsoft [43]. First, we briefly discuss the existing bug and its properties that are necessary for writing a query. We then illustrate the query for locating variants of this bug.

To discover the variants of CVE-2021-31979, we must understand its root cause. For this, we use IDA Hex-Rays to decompile `RtlpCreateServerAcl`, which contains vulnerable code for CVE-2021-31979. Figure 3a shows its simplified version. At a high level, this function copies the contents of `Aces` into a new memory region. More specifically, it calculates the total content size of `Aces` by enumerating them (Line 4–8). Then, it allocates the new memory according to the calculated size using `ExAllocatePoolWithTag` (Line 9). Finally, it enumerates `Aces` again to copy their contents into the new memory (Line 12–16). Because an integer overflow check is absent when computing the total size whose type size is 2 bytes, it can allocate smaller memory than the contents of `Aces`, leading to a heap overflow.

In summary, the heap overflow bug in Figure 3a has the following properties:

1) The allocation size of `ExAllocatePoolWithTag` is a 2-byte integer type and not a constant integer.
2) A feasible path exists from the function entry through `ExAllocatePoolWithTag` to `memcpy`.
3) The destination address of `memcpy` is derived from the allocation result of `ExAllocatePoolWithTag`. In addition, the copy size of `memcpy` can be greater than the allocation size of `ExAllocatePoolWithTag`.

Based on these properties, we can write a query to discover other heap overflow bugs, as shown in Figure 3b.

```
1  __int64 __fastcall RtlpCreateServerAcl(...) {
2    unsigned short AclSize = 8;
3    ...
4    if ((_WORD)AceCount) {
5      for (i = 0; i < AceCount; i++) {
6        AclSize += Aces[i]->AceSize;
7        ...
8    }
9    buffer = ExAllocatePoolWithTag(PagedPool, AclSize, 'cAeS');
10   ...
11   AclOffset = 8;
12   for (i = 0; i < AceCount; i++) {
13     ...
14     memcpy (buffer + AclOffset, Aces[i], Aces[i]->AceSize);
15     AclOffset = AclOffset + Aces[i]->AceSize;
16   }
17 }
```

(a) CVE-2021-31979 [47], a heap overflow bug example in the Windows kernel, especially `RtlpCreateServerAcl` of ntoskrnl.exe.

Writing query

```
function symRule (node) {
  if (isCall(node,"ExAllocatePoolWithTag")
    && node.args[1].type.size == 2
    && !isConstant(node.args[1])) {
    setCallback(node, function(node, state) {
      var addr = state.getValue(node);
      var size = state.getValue(node.args[1]);
      if (state.allocs == undefined) state.allocs = [];
      state.allocs.push({addr: addr, size: size, node: node});
    }, "malloc");
  }

  if (isCall(node, "memcpy")) {
    setCallback(node, function(node, state) {
      var dst = state.getValue(node.args[0]);
      var size = state.getValue(node.args[2]);
      for (var alloc of state.allocs) {
        if (dst.includes(alloc.addr)
          && state.isSAT(alloc.size < size)) {
          print("Overflow detected");
        }
      }
    }, "memcpy");
  }
  addDependency("malloc", "memcpy");
}

for (func in prog.functions)
  symExec(func, symRule);
```

(b) A query for finding heap overflow bugs.

Executing query

```
1  NTSTATUS __fastcall NtfsSetDispositionInfo(...) {
2    ...
3    length = volumeName->Length + dirName->Length
4             + fileName->Length;
5    path.Length = 0;
6    path.MaximumLength = length + 2;
7    path.Buffer = (PWSTR)ExAllocatePoolWithTag(
8                                  (POOL_TYPE)(PoolType | 0x10),
9                                  (unsigned __int16)(length + 2),
10                                 'FFtN');
11   RtlAppendUnicodeStringToString(&path, volumeName);
12   ...
13   memcpy(
14     (char *)path.Buffer + path.Length,
15     dirName->Buffer,
16     dirName->Length);
17   ...
18 }
```

(c) CVE-2021-41378, one of the heap overflow bugs found by QUERYX.

Figure 3: A running example of QUERYX for heap overflow bugs.

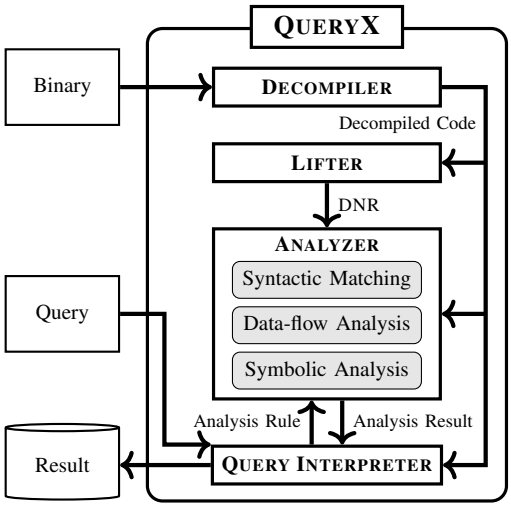


Figure 4: QUERYX Architecture.

This query starts under-constrained symbolic execution from all function entries in the program, based on the symbolic rule specified by `symRule` (Line 28–29). To verify the first property, the query sets a callback to the nodes that call `ExAllocatePoolWithTag` if its size argument is a 2-byte type and not constant (Line 2–4). The callback obtains the allocated memory and its size and then stores them in the symbolic state (Line 5–10). In addition, this query registers a callback for every node that calls `memcpy`. In Line 17–22, the callback reports a heap overflow if the destination address of the `memcpy` arguments includes the allocated address and the copy size of `memcpy` is greater than the size of the allocated address, which is true for the second and third properties. Notably, this query sets a dependency between `malloc` and `memcpy` callbacks (Line 25). This allows QUERYX to reduce the CFG for more scalable analysis, which is described in §5.2.

Finally, we applied the query shown in Figure 3b to the Windows kernel and found 7 unique bugs. Figure 3c describes one of the discovered heap overflow bugs with the simplified decompiled code of `NtfsSetDispositionInfo`. In summary, this function allocates a new memory region to create a full path by concatenating its volume, directory, and file names. Notably, this function additionally has the following properties: the allocation size of `ExAllocatePoolWithTag` is a 2-byte integer type and the copy size of `memcpy` can be greater than the allocation size of the destination buffer due to integer overflow. Therefore, QUERYX discovered this bug, which leads to a heap overflow, based on the query.

3.2. Architecture

Figure 4 depicts the architecture of QUERYX. At a high level, QUERYX produces analysis results by analyzing a given binary code based on a query written by analysts. To accomplish this, QUERYX consists of four major modules: DECOMPILER, LIFTER, ANALYZER, and QUERY INTERPRETER. The DECOMPILER module first decompiles the binary code using the existing binary decompiler, IDA Hex-Rays [28]. And the LIFTER module transforms the decompiled code into our IR called DNR for the ANALYZER module. Lastly, the QUERY INTERPRETER module evaluates the given query, instructs ANALYZER on which analysis to perform, and receives the analysis results.

LIFTER. This module transforms the decompiled code into DNR. For precise analysis, QUERYX considers binary-dependent code (e.g., offset-based memory access), binary-embedded data (e.g., global variables), and decompiled-induced code (e.g., LOBYTE macro in IDA Hex-Rays). To simplify this complicated information, we design our IR named DNR and convert the decompiled code into DNR. We also maintain a record of which AST node in the decompiled code is converted into which IR to facilitate the analysis on the AST. This lifting only uses decompiled code as its input, thereby making QUERYX available even with the closed-source decompiler, IDA Hex-Rays (see §4).

ANALYZER. The ANALYZER module takes DNR from the LIFTER module, decompiled code, and analysis rules, which are specified in the given query, and returns the analysis result. This module supports three analysis techniques: syntactic matching, data-flow analysis, and symbolic analysis. As with existing tools (e.g., CodeQL and joern), the syntactic matching is performed by traversing the AST of the decompiled code based on the given query. The data-flow analysis is performed by specifying the source and destination nodes and checking whether flow exists between them. In the case of symbolic analysis, the query checks or changes symbolic states by adding callbacks to AST nodes that the analysts want to check or change. This symbolic analysis is inherently equipped with under-constrained symbolic execution and CFG reduction, making the analysis scalable (see §5). Notably, in this paper, we focus on symbolic analysis, which is the most interesting part of QUERYX.

QUERY INTERPRETER. This module parses and executes a given query by interacting with the ANALYZER module. The query specifies which and how the analysis will be used by the ANALYZER module and retrieves the analysis results. And the QUERY INTERPRETER module offers JavaScript-like language to alleviate the learning curve associated with query writing. To make this step intuitive, this module enables analysts to write a query based on the AST of the decompiled code. This module also provides several handy interfaces for analysis; e.g., this allows checking AST types and value types of AST expressions for callback registration.

4. Decompiler-Neutral Representation (DNR)

Although decompiled code contains high-level information and is intuitive to analysts, the AST of decompiled code is inconvenient for implementing analysis, as discussed in §2.2. Therefore, we design an analyzer-friendly and decompiler-neutral representation, named DNR. This simplifies the analysis of QUERYX while preserving the high-level information of the decompiled code. In this section, we describe the structure of DNR and the process of lifting DNR from the decompiled code.

4.1. DNR Structure

DNR expresses a program as a set of functions and program data, which includes binary-embedded data such as global variables (see §A). A function consists of a name, an offset in which the function is located in the program, arguments, and a body. The body is a sequence of basic blocks, each of which is composed of statements. Similar to other IRs, DNR statements represent standalone units of executions, such as variable definitions, memory allocations, stores, and control flow changes, whereas DNR expressions represent values. DNR has three characteristics for preserving the high-level information of decompiled code and semantics of the program: metadata, program address, and program data.

Metadata. Many IRs have their metadata, which is extra information related to the corresponding IR, such as branch type information, analysis results, and a hash value for hash-consing [9, 31, 56]. Similarly, all DNR expressions can have two types of metadata: expression information (`ExprInfo`) and AST information (`AstInfo`). `ExprInfo` has expression-specific information, such as a hash value for hash-consing, and `AstInfo` points to which AST node is lifted to the corresponding expression. Notably, the expression AST node contains high-level information of decompiled code, such as syntactic information (e.g., a kind of AST node, children nodes, a parent node, and location) and value types. Therefore, our symbolic analysis allows analysts to specify symbolic rules based on the AST of decompiled code using the AST information in metadata, as explained in §5.1.

Program Address. Global variables in the decompiled code have names (e.g., `dword_2010A4`) and locations (e.g., `0x2010A4`). If the binary code is compiled as a position-independent code, which is common for achieving Address Space Layout Randomization (ASLR), their locations should be distinguished from absolute addresses. We thus explicitly express their locations, which are offsets from the base address of the program, by employing `ProgramAddr`. For example, QUERYX lifts reading the global variable, `dword_2010A4` in Figure 2, into `Load(ProgramAddr(0x2010A4),4)`.

Program Data. Decompiled code includes code that accesses program data such as global variables. To fully express the semantics of accessing program data, QUERYX divides all program data into sections and records their ranges, permissions, and initial values. This information is used to properly handle memory accesses in QUERYX. Additionally, QUERYX considers the relocation of the program data. This is crucial for correctly capturing the runtime semantics of a particular value. For example, while the value of a certain location was just `0x1234`, its actual value during the runtime could be `ProgramAddr(0x1234)` due to the relocation.

4.2. DNR Lifting

The major parts of lifting decompiled code into DNR are similar to lifting the source code into ordinary IR (e.g., LLVM IR). However, QUERYX considers several issues to preserve the semantics of the program.

Local Variables. In decompiled code, local variables are stored in either stacks or registers, and their locations can affect the semantics of the decompiled code, as discussed in §2.2. Lifting local variables in registers into variables of DNR is straightforward; we use a dedicated variable for each register. For local variables in stacks, we obtain their stack offsets from the decompiler. Then, we calculate the total stack size from this information and allocate the stack with an `Alloc` statement. Finally, we convert accessing local variables in the stack into memory operations (i.e., `Store` and `Load`) with the allocated stack and the corresponding stack offsets.

Array Variables. Decompiled code has diverse types of variables, such as an integer, a pointer, and an array. Particularly, an array-type variable indicates the address where its elements are stored. Therefore, we lift array-type variables into the addresses where they point to. For example, QUERYX lifts `unk_0x1234` into `ProgramAddr(0x1234)` if `unk_0x1234` is an array and lifts `dword_0x5678` into `Load(ProgramAddr(0x5678),4)` if `dword_0x5678` is not an array, such as `DWORD`.

Decompiled-induced Code. To simplify the decompiled code, the decompiler uses macros that have the same semantics as that of the original code without explicitly defining these macros. We refer to such macros as decompiler-induced code. This simplification can be intuitive for analysts but not for analyzers because of its implicitness. Therefore, we resolve macros into statements or expressions of DNR while preserving their semantics. For example, `LOBYTE(X)` in Figure 2, which returns the lowest byte of value `X`, is lifted as `Extract(X,0,1)`. This implies extracting `1` byte from the `0` index of `X`. For now, we support macros related to extracting values, such as `LOBYTE`, `LOWORD`, `LODWORD`, `HIBYTE`, `HIWORD`, `HIDWORD`, `BYTE1-15`, `WORD1-7`, and `DWORD1-3`.

5. Design

In this section, we present the design of QUERYX about how QUERYX performs symbolic analysis on DNR with queries based on decompiled code (§5.1), CFG reduction by callbacks and their dependencies for the scalability of symbolic analysis (§5.2), and other details of symbolic analysis (§5.3).

5.1. Symbolic Analysis

Under-constrained Symbolic Analysis. For the scalability, QUERYX begins the symbolic analysis from function entries instead of the program entry as under-constrained symbolic execution of UC-KLEE [52]. In particular, QUERYX analyzes functions with unknown (i.e., under-constrained) values; QUERYX treats function arguments or memory values as unknown values if they cannot be determined during analysis. Despite this limited environment, under-constrained symbolic analysis can produce meaningful results from the relationships between unknown values (e.g., `unknownX > unknownY`) or constant expressions (e.g., `unknownZ > 256`).

Binary-aware Analysis. QUERYX employs binary-aware analysis to ensure precise analysis. First, QUERYX reflects the binary runtime environment in symbolic memory. More specifically, QUERYX divides the memory into three regions: stack, program data, and an unknown region. Similar to machine memory, QUERYX uses a flat array to represent the stack for each function. Then, QUERYX arranges the variables according to their layout, as we described in §4.2. QUERYX defines the program data region as a single flat array and loads values lazily from the binary. And QUERYX uses unknown regions to represent unknown objects in under-constrained symbolic analysis. Similar to UC-KLEE [52], QUERYX assumes that unknown references point to unique objects due to the absence of aliasing information. Second, QUERYX deals with memory access differently according to their permissions in the binary code. When QUERYX reads a value from a global variable, QUERYX uses an unknown value for writable memory, but a corresponding initial value is used for read-only memory.

Callback-based Analysis. QUERYX suggests callback-based analysis to achieve intuitive query writing; analysts can register a callback for an AST node to control symbolic analysis while accompanying the execution flows of the target program. To implement this, when QUERYX completes symbolically executing a DNR expression, QUERYX checks whether the AST node in the metadata of the DNR expression contains callbacks. If the callback is present, QUERYX invokes the callback based on the current symbolic state.

5.2. CFG Reduction

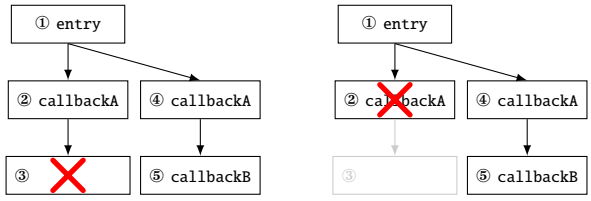
Despite performing under-constrained symbolic analysis, QUERYX still suffers from the scalability issue when analyzing large COTS binaries (e.g., Windows kernel). To overcome this issue, QUERYX reduces its CFG before starting the symbolic analysis. Thanks to our callback-based query model, CFG can easily be reduced by checking for the presence of callbacks. Additionally, QUERYX allows analysts to define ordering dependencies between callbacks. These are straightforward to define (e.g., a few lines) and offer more opportunities for CFG reduction. Notably, such ordering dependencies are natural in discovering vulnerabilities. For example, to discover heap overflow, we must analyze paths that contain ordering dependencies between the memory allocation (e.g., `malloc`) and the memory copy (e.g., `memcpy`).

Algorithm 1 illustrates the pseudocode for CFG reduction of QUERYX. The `reduceCFG` function takes a CFG denoted as G, and a set of ordering dependencies specified by the query, which is denoted as D. Then, this function outputs the reduced CFG. At a high level, the `reduceCFG` function removes nodes and edges that are irrelevant to callbacks by calling `reduceByCallback`, then trims the reduced CFG by ordering dependencies.

CFG Reduction by Callbacks. In Line 16 of Algorithm 1, the `reduceByCallback` function deletes the nodes and edges that are disconnected from nodes with callbacks. This function first obtains a set of the entry node (N_{entry}) and collects

Algorithm 1: CFG Reduction

```
Input  : A function CFG, G
         A set of ordering dependencies, D
Output : A reduced CFG
1  function reduceByCallback(G)
2      N_entry ← getEntryNode(G)
3      N_cb ← filterNodes(G, hasCallback)
4      return getSubCFG(G, N_entry, N_cb)
5  function reduceByDependency(G, d)
6      G' ← G_empty
7      N_cur ← getEntryNode(G)
8      for i ← 0 to d.Length do
9          N_nxt ← getCallbackNodes(G, d[i])
10         G_sub ← getSubCFG(G, N_cur, N_nxt)
11         if G_sub = G_empty then return G_empty
12         G' ← mergeCFG(G', G_sub)
13         N_cur ← N_nxt
14     return G'
15 function reduceCFG(G, D)
16     G ← reduceByCallback(G)
17     G' ← G_empty
18     for d ∈ D do
19         G_tmp ← reduceByDependency(G, d)
20         G' ← mergeCFG(G', G_tmp)
21     return G'
```



(a) CFG reduction by callbacks. **(b)** CFG reduction by dependencies.

Figure 5: An example of CFG reduction where there is an ordering dependency that `callbackB` must be called after `callbackA`.

nodes with callbacks (N_{cb}). This function then calculates and returns the sub-CFG that starts at the nodes in N_{entry} and ends at the nodes in N_{cb}. For example, node ③ in Figure 5a is removed because no callback is present after this node. The nodes without callbacks are uninteresting so this reduction effectively eliminates uninteresting paths.

CFG Reduction by Dependencies. `reduceByDependency` takes a CFG denoted as G, and an ordering dependency, which is a sequence of callback names, denoted as d, and returns the reduced CFG. This function gets the next nodes (N_{nxt}) by the dependency in Line 9 and calculates the sub-CFG that starts at the nodes in N_{cur} and ends at the nodes in N_{nxt}. The sub-CFG then merges with the resulting CFG, which is initialized as an empty CFG, unless the sub-CFG is empty. Next, d is iterated and this process is repeated. Notably, N_{cur} in the first iteration includes only the entry node of G, and N_{cur} is updated as N_{nxt} after merging the CFGs. Finally, this function returns the reduced CFG, which has no nodes or edges that do not satisfy the dependency. For instance, if the query specifies that `callbackB` precedes `callbackA`, node ② in Figure 5b is removed because no node after node ② has `callbackB`.

Idx	CVE	Program	Function	Bug Type	Impact	Bounty
1	CVE-2021-41370	ntfs.sys	NtfsSetShortNameInfo	Heap overflow	Elevation of Privilege	$20,000
2	CVE-2021-41378	ntfs.sys	NtfsSetDispositionInfo	Heap overflow	Remote Code Execution	$20,000
3	CVE-2021-43229	ntfs.sys	TxfAllocateAndStoreNameForTxfLogging	Heap overflow	Elevation of Privilege	$20,000
4	CVE-2021-43230	ntfs.sys	TxfAllocateFullFilePathForChangeNotify	Heap overflow	Elevation of Privilege	$20,000
5	CVE-2021-43231	ntfs.sys	NtfsRenameToPrivateDir	Heap overflow	Elevation of Privilege	$20,000
6	CVE-2021-41367	ntfs.sys	TxfOpenFileProcessing	Heap overflow	Elevation of Privilege	$20,000
7	CVE-2022-23293	fastfat.sys	FatSetFullNameInFcb	Heap overflow	Elevation of Privilege	$20,000
8	CVE-2022-30162	win32kfull.sys	NtUserSetClassLongPtr	Kernel Address Disclosure	Information Disclosure	-
9	CVE-2020-17041	PrintConfig.dll	CopyFileFromPrinterData	Path Traversal	Elevation of Privilege	$20,000
10	CVE-2020-17042	PrintConfig.dll	UniDrvUI::PConcatFilename	Path Traversal	Remote Code Execution	$20,000
11 – 15	-	Automotive	REDACTED	Out-of-bound Access	Elevation of Privilege	-

TABLE 2: Unique bugs found by QUERYX.

Query	QUERYX	angr	joern
Heap Overflow	38	267	12
Kernel Address Disclosure	65	233	10
Path Traversal	74	248	13
OOB Access	113	341	-

TABLE 3: Lines of queries of each tool for four types of bugs.

Alternatives to Path Slicing. For usability, QUERYX intentionally chose this CFG reduction over path slicing [12, 30, 55], which is a widely used technique for scalable symbolic execution. If we can specify all the critical points of a path (i.e., targets), path slicing can outperform the proposed CFG reduction because path slicing can even eliminate nodes that are not at the end of the path. Unfortunately, callbacks are necessary but insufficient to represent the targets of a path. In QUERYX, analysts can freely use symbolic states for analysis (e.g., constraints or symbolic memory). Therefore, any symbolic state access should be considered as a target in path slicing, which makes path slicing inefficient due to its large number. One method to avoid this is to allow analysts to annotate the symbolic state that they will use. However, we believe that this could impair the usability of QUERYX. Despite the current CFG reduction, we showed that QUERYX can be sufficiently scalable to handle large COTS binaries in §8.3. Therefore, QUERYX employs this simple yet conservative CFG reduction, instead of sophisticated path slicing.

5.3. Other Details of Symbolic Analysis

Loop Unrolling. One of the main reasons for the path explosion in the symbolic analysis is the loop in the program. To mitigate being stuck within loops, QUERYX limits the number of iterations in each loop. This is also known as loop unrolling, and many analyzers [40, 67, 72, 81] have already employed this. Notably, the maximum number of iterations is set by a query, and the default number is two.

Inter-procedural Analysis. We support inter-procedural analysis but limit the maximum depth of function calls to mitigate the path explosion. Similar to the loop count, this value is set by a query and the default number is one. This means that QUERYX performs an intra-procedural analysis by default. And analysts can explicitly instruct QUERYX to follow a certain function using `state.symExec`. In §7.3, we demonstrate the inter-procedural analysis of QUERYX.

6. Implementation

We have implemented QUERYX with 1.6K lines of Python code, 40 lines of JavaScript code, and 9.6K lines of F# code. Specifically, the DECOMPILER module employs IDA Hex-Rays [28] to decompile binary code and dumps the decompiled code using 1.6K lines of IDA Python code. And the QUERY INTERPRETER module is implemented with 2.4K lines of F# code and 40 lines of JavaScript code while it imports Esprima [29], which is a Node.js library, for parsing JavaScript-like queries. In the ANALYZER module, we re-implemented an under-constrained symbolic execution for DNR (§5.1), which UC-KLEE [52] originally suggested for LLVM IR, and employed Z3 [48] version 4.8.10 to solve path constraints. Notably, the LIFTER and ANALYZER modules consist of 3.0K and 4.2K lines of F# code, respectively.

7. Using QUERYX to Find Bugs

In this section, we apply QUERYX to find several bugs and show that QUERYX has the following properties.

To demonstrate that QUERYX can find diverse types of bugs (**Expressiveness**), we implemented queries for four types of bugs: heap overflow, kernel address disclosure, path traversal, and out-of-bound access. Table 3 shows the lines for each query. In the following subsections, we discuss each query for finding heap overflow, kernel address disclosure, and out-of-bound access. We include the query for path traversal in Appendix §C due to space limits.

To show that QUERYX can discover bugs in binaries with different architectures (**Applicability**), we ran the queries on the Windows kernel and Windows system service, which are based on x64, and an automotive binary based on ARM. Table 7 depicts the version, size, and decompilation time of the target binaries. Note that we used a public symbol server for Windows binaries and debugging symbols in the automotive binary to help decompilation.

Finally, we found 15 previously unknown bugs including 10 CVEs, and earned $180,000 in bounty, as shown in Table 2. These results highlight that QUERYX is effective in finding bugs in real-world binaries (**Effectiveness**). And Table 4 shows the detailed results while we considered our findings as true positives only if we could generate proof-of-concepts for them. Notably, the time in Table 4 represents the analysis time without the decompilation time.

Query	Binary	QUERYX				angr				joern			
		Time	Total	TP	FP	Time	Total	TP	FP	Time	Total	TP	FP
Heap Overflow	ntfs.sys	3.9h	39	6	33	6.5h	39 (26)	3 (3)	36 (23)	2.3h	126 (32)	5 (5)	121 (27)
	fastfat.sys	16.6m	6	1	5	20m	6 (4)	1 (1)	6 (3)	27m	30 (6)	1 (1)	29 (5)
	win32kfull.sys	1.3m	2	0	2	2m	2 (2)	0	2 (2)	6.1h	5 (2)	0	5 (2)
Kernel Address Disclosure	ntfs.sys	50s	0	0	0	54m	0	0	0	2.2h	0	0	0
	fastfat.sys	13s	0	0	0	3.5m	0	0	0	29m	0	0	0
	win32kfull.sys	2.1h	3	1	2	7.5h	2 (2)	1 (1)	1 (1)	6.7h	3 (3)	1 (1)	2 (2)
Path Traversal	PrintConfig.dll	1.5m	8	2	6	38m	8 (8)	2 (2)	6 (6)	4.3h	26 (8)	2 (2)	24 (6)
OOB Access	Automotive	4.1m	5	5	0	1h	5 (5)	5 (5)	0	-	-	-	-

TP: True positive. **FP**: False positives. The numbers in the parentheses are the number of cases that QUERYX also found.

TABLE 4: Query results of QUERYX, angr, and joern on the Windows kernel, Windows system service, and an automotive binary.

Experimental Setup. We evaluated QUERYX on a machine with an AMD Ryzen 3900X (12 cores) and 112GB RAM, running 64-bit Ubuntu 18.04 LTS. And we used the QUERYX default configuration: 24 hours for each program's timeout, 10 min for each function's timeout, 2 for the maximum loop iteration, 1 for the maximum depth of function calls.

7.1. Heap Overflow

To find heap overflow bugs, we wrote a query as described in §3.1. We then applied the query to a part of the Windows kernel: ntfs.sys, fastfat.sys, and win32kfull.sys.

Result. Table 4 shows that QUERYX found 47 heap overflow bug candidates in the three binaries: 39 candidates in ntfs.sys, 6 candidates in fastfat.sys, and 2 candidates in win32kfull.sys. By manual analysis, we concluded 6 bugs and 33 false positives in ntfs.sys, 1 bug and 5 false positives in fastfat.sys, and 2 false positives in win32kfull.sys. Table 2 lists heap overflow bugs found by QUERYX and §3.1 describes one of them. Furthermore, it took 3.9 hours, 16.6 min, and 1.3 min to analyze ntfs.sys, fastfat.sys, and win32kfull.sys, respectively.

False Positives. The false positives can be attributed to three main reasons. (1) QUERYX employs under-constrained symbolic execution, which begins with function entries. QUERYX assumes that all unknown values, such as function arguments, can have arbitrary values even though they are not. For example, QUERYX reports `FormFullImageName` in win32kfull.sys as a heap overflow candidate based on the assumption that its allocation size can be overflowed. However, we found that the only caller of the function specifies the size as constant, thereby making overflow impossible. (2) Due to the absence of a complete list of exit functions, QUERYX passed them instead of stopping, thereby resulting in false positives. For instance, `NtfsCacheSharedSecurityBySecurityId` calls an exit function if integer overflow occurs. However, QUERYX cannot recognize this exit function and incorrectly concludes that integer overflow is possible. (3) Finally, QUERYX discovered some bug candidates such as `NtfsProcessRepairVerbIndexEntry` that cannot be triggered in our threat model. This function can only be invoked when repairing NTFS file systems. Unfortunately, in Windows, only the administrator can repair file systems. Therefore, these candidates were non-security bugs and false positives.

7.2. Kernel Address Disclosure

We also implemented a query for finding kernel address disclosure bugs and evaluated this query on three Windows kernel binaries, which are the same as in §7.1. Note that such kernel address disclosure bugs are essential in modern exploitations to bypass one of the common exploit mitigations, kernel address space layout randomization (kASLR) [18].

Query. The Windows kernel usually writes values into the user-space memory to transfer them into user-space. We thus wrote a query for kernel address disclosure bugs in which kernel addresses are stored in the user-space memory, as shown in Figure 6a. At a high level, we collect user-space and kernel addresses then check whether the kernel addresses are stored in the memory pointed by the user-space addresses. We determine user-space addresses based on the fact that the Windows kernel usually checks whether an address points to the user-space by calling `ProbeForWrite` or comparing it with `MmUserProbeAddress` before writing a value to the user-space memory (Line 1–6, 10–15). And we collect all addresses by checking whether they are pointer types, which can be kernel or user-space addresses (Line 17–22). Finally, when an AST node writes a pointer-typed value into a memory address (e.g., `*x = y` or `x->a = y` or `x[i] = y`), we conclude this as a kernel address disclosure bug if the address is a user-space address and the value is a kernel address (Line 24–34). Notably, `isUserAddr(state, addr)` and `isPtr(state, addr)` return true if `addr` is one of the values added by `addUserAddr` and `addPtr`, respectively.

Result. After evaluating the query on the three binaries, QUERYX found 3 kernel address disclosure bug candidates in only win32kfull.sys. In addition, it took 50s, 13s, and 2.1h to analyze ntfs.sys, fastfat.sys, and win32kfull.sys, respectively. ntfs.sys and fastfat.sys took less time and had no bug candidates because they implement file systems that transfer values to the user-space less frequently. We determined a genuine bug and two false positives by manual investigation.

Figure 6b shows the kernel address disclosure bug found by QUERYX. In summary, `ptr` is set to `&buf`, which is a kernel stack address, in Line 7 and stored in the user-space memory in Line 13. Specifically, QUERYX with the query first adds `&buf` to a set of pointers after executing Line 7. Next, `user_addr` is added to a set of user-space addresses

```
function isUserAddrSrc(node) {
  // Returns true if the node is
  //  1) the first argument of ProbeForWrite,
  //  or 2) compared with MmUserProbeAddress
  ...
}

// Goal: Find *userAddr = kernelPtr;
function symRule(node) {
  if (isUserAddrSrc(node)) {
    setCallback(node, function (node, state) {
      var userAddr = state.getValue(node);
      addUserAddr(state, userAddr);
    })
  }

  if (isPtr(node)) {
    setCallback(node, function (node, state) {
      var ptr = state.getValue(node);
      addPtr(state, ptr);
    })
  }

  if (isMemWrite(node) && isPtrType(node.value)) {
    setCallback(node, function (node, state) {
      var addr = state.getValue(getMemAddr(node.assignee));
      var value = state.getValue(node.value);
      if (isUserAddr(state, addr)
        && isPtr(state, value)
        && !isUserAddr(state, value)) {
        print("Kernel address disclousre detected");
      }
    });
  }
}
```

(a) A query for finding kernel address disclosure bugs.

```
_QWORD *__fastcall NtUserSetClassLongPtr(...) {
  ...
  __int128 *ptr; // [rsp+48h] [rbp-70h]
  ULONG64 v26; // [rsp+58h] [rbp-60h]
  __int128 buf; // [rsp+60h] [rbp-58h]
  ...
  ptr = &buf;
  v8 = xxxSetClassLongPtr(v11, -8, (__int64)&v24, a4);
  v21 = MmUserProbeAddress;
  if ( user_addr >= MmUserProbeAddress )
    user_addr = MmUserProbeAddress;
  *(_OWORD *)user_addr = v24;
  *(_QWORD *)(user_addr + 16) = ptr;
  ...
}
```

(b) CVE-2022-30162, a kernel address disclosure bug QUERYX found.

Figure 6: A query for finding kernel address disclosure bugs and a bug found by QUERYX.

after Line 10. Finally, in Line 13, this code is concluded as a bug after validating that the memory address for write is a user-space address and the value for write is a kernel address (i.e., an address that is not a user-space address).

False Positives. False positives occur from this query for two specific reasons. First, we tried to determine which address points to the user-space memory using `ProbeForWrite` and `MmUserProbeAddress`, but this was incomplete. For example, `NtGdiSTROBJ_bEnumInternal` has a user-space address defined by another method that we did not notify. Therefore, the query confused the pointer with a kernel address and classified this as a bug. Second, we assumed that a pointer typed value would be an address, but there was an exception. In `NtUserGetClipboardAccessToken`, the decompiler analyzed the output of the `ObOpenObjectByPointer` function as a pointer type, but it was actually a handle type. This broke our assumption and caused a false positive.

7.3. Out-of-Bound Access

In this subsection, we illustrate our query for discovering out-of-bound (OOB) accesses in an automotive binary. This example differs from the previous ones in three ways. First, the automotive binary is based on ARM, not x86, which demonstrates the applicability of QUERYX. Second, this binary contains several indirect calls based on virtual function tables (vtable). Without resolving indirect call targets, QUERYX cannot perform meaningful analysis. Finally, this binary supports IPC services based on an Android binder, the handlers of which have names such as `Service::onTransact`. We found that such handlers typically invoke other functions to process their inputs. Therefore, we need an inter-procedural analysis instead of an intra-procedural analysis.

Our key idea is to discover OOB accesses by checking whether free (i.e., unconstrained) inputs are used as memory addresses. In particular, we first assume that we can fully control the return values from Android parcel functions such as `android::Parcel::readInt32`. Then, we attempted to find the memory access dependent on such values but not restricted (e.g., without boundary checks). If such memory access exists, we conclude that it is OOB access.

Query. A simplified query for finding OOB access is shown in Figure 7a. QUERYX resolves indirect call targets based on vtables. In particular, we write a vtable address into the memory pointed to by `this` object, as vtable is typically located at the start of the object (Line 9–16). Then, we add user-controllable inputs into the symbolic state, which are the return values of the Android parcel functions (Line 18–22). For efficient inter-procedural analysis, we analyze the callee only if one of its arguments has free inputs (Line 24 – 36). Finally, we concluded that they are OOB access bugs if memory addresses have any free inputs (Line 38–44).

Result. QUERYX took 4.1 minutes to analyze the target binary and found 5 bug candidates. We manually analyzed them and confirmed that all of these candidates were bugs. We believe that this level of precision can be achieved because this query incorporates extensive domain knowledge (e.g., sources of user-controllable values and vtables) and more in-depth analysis (e.g., inter-procedural analysis). Figure 7b illustrates one of the found bugs with anonymized names. `Service::onTransact` reads values from the IPC message in Line 7–8 and passes them into the indirect call, which target is resolved as `Serivce::vuln`. In Line 16, an input is used to calculate the memory address for the write without any checks. Therefore, this leads to an OOB access bug.

8. Evaluation

In this section, we evaluate QUERYX by comparing it with the state-of-the-art extensible static checking tools, such as angr, joern, Sys, and CodeQL (§8.1, §8.2) in terms of query writing and bug findings. Due to the space limit, we include our evaluation in the dataset from Mantovani et al [42] in Appendix §E. We also demonstrate how CFG reduction affects the scalability of QUERYX based on the existence of callbacks and dependencies (§8.3).

```
function hasFreeInput(state, value) {
  // Return true if value depends on
  // an input that has no constraint.
  ...
}

// Goal: Find OOB accesses
function symRule (node) {
  if (isThis(node)) {
    setCallback(node, function (node, state) {
      var thisAddr = state.getValue(node);
      var vtable = getVtable(state);
      if (vtable != undefined)
        state.writeMem(thisAddr, vtable, 8);
    });
  }

  if (isReadInput(node)) {
    setCallback(node, function (node, state) {
      addInput(state, state.getValue(node));
    });
  }

  if (isFuncCall(node)) {
    setCallback(node, function (node, state) {
      var args = state.getValues(node.args);
      for (var arg of args) {
        if (hasFreeInput(state, arg)) {
          var callee = state.getValue(node.callee);
          var calleeName = getFuncName(callee);
          state.symExec(calleeName, symRule, args);
          return;
        }
      }
    });
  }

  if (isMemAccess(node)){
    setCallback(node, function (node, state) {
      var addr = state.getValue(getMemAddr(node));
      if (hasFreeInput(state, addr))
        print("Out-of-bound detected");
    });
  }
}
```

(a) A query for out-of-bound access bugs.

```
__int64 Service::onTransact(Service *this, ...,
                            const android::Parcel *a3, ...) {
  ...
  switch (...) {
    case 0x40:
      ...
      input1 = android::Parcel::readInt32(a3);
      input2 = android::Parcel::readInt32(a3);
      // Will be resolved to Service::vuln
      (*(*(_QWORD *)this + 0x40LL))(this, input1, input2);
      ...
}

__int64 Service::vuln(Service *this, int a1, int a2) {
  ...
  *((_DWORD *)this + a2 + 0x40) = a1;
  ...
}
```

(b) One of the out-of-bound access bugs found by QUERYX.

Figure 7: A query for finding out-of-bound access bugs and one of the bugs found by QUERYX.

8.1. Comparison against angr and joern

For comparison, we wrote queries for angr and joern to discover four types of bugs that QUERYX had already covered in §7. We made every effort to match the query semantics with those of QUERYX. Note that we used queries with data-flow analysis for joern because it does not support a symbolic query. Finally, using the same setup as §7, we ran angr on binaries and joern on the decompiled code.

vs angr. Theoretically, angr, as a general symbolic analysis platform, should have capabilities that are similar to those of QUERYX. However, as shown in Table 4, angr missed 3 true bugs that QUERYX discovered and took 1.6× more time on average. QUERYX outperformed angr due to two main reasons: simpler and more intuitive query writing than that of angr and different memory models.

According to our experience, query writing of QUERYX is simpler and more intuitive than that of angr. Table 3 shows that QUERYX requires fewer lines of queries than angr. This is because the decompiled code has analyst-friendly information and advanced analyses, thereby allowing QUERYX to employ diverse optimizations. Even with our efforts, we could not apply the same optimizations to angr because it involves substantial engineering work.

For example, during kernel address disclosure bug detection, QUERYX uses value types in the decompiled code to find pointers and enable more aggressive path pruning. Unfortunately, angr cannot apply this optimization due to the lack of such information. Moreover, we discovered that conditional constant propagation in IDA Pro allows QUERYX to filter out uninteresting cases in advance (e.g., `memcpy` with a conditionally constant size) when detecting heap overflow. In contrast, angr limits such an aggressive analysis because it aims to accurately represent the semantics of code blocks from a binary [17]. During OOB detection, we filtered out local variable writes using the fact that memory writes for local variables can be easily distinguished in decompiled code: local variable writes are represented as assignments rather than dereferences. However, angr requires additional analysis to achieve that. We thus made angr to monitor every memory access, which slowed down its analysis.

In addition, QUERYX employs different memory models from angr. While QUERYX follows the memory models of UC-KLEE, in which unknown addresses point to unique objects, angr assumes that they can point to the same object. As a result, even though it is impossible, angr sometimes concludes that a write to an unknown address overwrites an object pointed to by another unknown address. This causes differences between QUERYX and angr in heap overflow detection, as shown in Table 4.

vs joern. joern supports data-flow analysis but not symbolic analysis. We thus wrote queries with data-flow analysis to find bugs in §7. Notably, joern cannot find OOB bugs because they require symbolic analysis. During our evaluation, we found that if joern observes a parsing error and a data-flow analysis error for a certain function, it skips a group of functions instead of only the problematic one. This can lead to unintentional false negatives in joern. Therefore, we ran queries on each function instead of the entire program for a fair comparison, and Table 4 shows the results. joern found all the bug candidates found by QUERYX except 7 heap overflow bug candidates in ntfs.sys. This is due to the inability of joern to monitor data flows through memory rather than variables. And due to the absence of symbolic analysis, joern results in 3.8× more false positives than QUERYX. This highlights that QUERYX outperforms joern in leveraging the decompiled code to find bugs.

System	Time	Total	TP	FP	Unknown
QUERYX	53s	35	22	8	5
Sys	40s	15	12	2	1
QUERYX ∩ Sys	–	14	12	1	1

TABLE 5: The result of finding heap out-of-bound access bugs in SQLite by QUERYX and Sys.

8.2. Comparison against Sys

Both QUERYX and Sys [7] are extensible static checking tools based on symbolic analysis. However, two key differences exist between them. First, QUERYX can analyze binary code based on decompiled code, whereas Sys requires a source code. Second, Sys requires two non-trivial queries for static and symbolic analyses. Meanwhile, QUERYX requires only one query, thanks to its fearless design. We showed these differences with the example of finding heap OOB access bugs by QUERYX and Sys.

Experimental Setup. To compare QUERYX and Sys, we used SQLite in Chrome commit `0163ca1bd8da`, which the authors of Sys evaluated in their paper. And we determined unique bugs by considering that multiple bug candidates can have the same root cause as the Sys paper. For the remaining experimental setups, we employed the same setup as in §7.

Query. For Sys, we used the query in the Sys paper, which is publicly available in its repository [8]. For QUERYX, we made two changes to the query in Figure 3b. First, we modified the query to determine memory allocation functions if function names end with "alloc". Second, we added semantics for OOB checks as Sys (e.g., boundary checks for array accesses). Notably, the queries for QUERYX and Sys were written in 48 and 335 lines of code, respectively. This highlights the simplicity of query writing in QUERYX.

Result. Table 5 presents the results of QUERYX and Sys for finding heap OOB bugs in SQLite. This demonstrates that QUERYX outperforms Sys in terms of bug detection. In particular, QUERYX found 10 more true bugs than Sys; QUERYX found 22 true bugs among 35 candidates while Sys found 12 bugs among 15 candidates. Notably, Sys produced fewer false positives because its static analysis could eliminate potentially erroneous candidates prior to performing symbolic analysis. QUERYX discovered all bug candidates that Sys found except for one. This is because IDA Hex-Rays incorrectly inferred an array expression as `idx[array]`, rather than as `array[idx]`. Unlike source code, such expressions can be ambiguous in a binary.

Appendix §D depicts one of the bugs that QUERYX uniquely found. The Sys query missed this because it failed to cover all LLVM IRs even though it was 6× longer than QUERYX query. This demonstrates that query writing in QUERYX is simpler and more intuitive than that in Sys.

8.3. Effectiveness of CFG Reduction

To show the effectiveness of CFG reduction, we evaluated three versions of QUERYX: QUERYX, QUERYX with CFG reduction by only callbacks and no dependencies (QUERYX$_{cb}$), and QUERYX without any CFG reduction (QUERYX$_{no}$). We ran them on three Windows kernel binaries with the heap overflow query in §7.1 because this query found several bugs and took a long time to analyze, thereby making it suitable for evaluating the effectiveness of the CFG reduction. Notably, we employed the same experimental setup as in §7 except the timeout of each program was unlimited.

Binary	QUERYX		QUERYX$_{cb}$		QUERYX$_{no}$	
	Time	TP	Time	TP	Time	TP
ntfs.sys	3.9h	6	7.75h	5	38.4h	5
fastfat.sys	16.6m	1	50m	1	3.1h	1
win32kfull.sys	1.3m	0	2.1h	0	39.5h	0

TABLE 6: The result of finding heap overflow bugs in three Windows kernel binaries by QUERYX, QUERYX with CFG reduction by only callbacks and no dependencies (QUERYX$_{cb}$), and QUERYX without CFG reduction (QUERYX$_{no}$).

Table 6 summarizes the results. Compared with QUERYX$_{cb}$ and QUERYX$_{no}$, QUERYX required the least time for analysis. Specifically, QUERYX$_{cb}$ took 1.5× longer than QUERYX because QUERYX$_{cb}$ must explore every function that contains `ExAllocatePoolWithTag` or `memcpy` without considering their dependencies. And QUERYX$_{no}$ took 18.3× longer than QUERYX because QUERYX$_{no}$ needs to traverse all nodes in all functions. More interestingly, QUERYX found one more true bug than QUERYX$_{cb}$ and QUERYX$_{no}$, as they reached timeout before finding it. This highlights that CFG reduction by callbacks and their dependencies is crucial for the scalability of QUERYX.

9. Discussion

Depending on Decompilers. Decompiled code enables QUERYX to support closed-source binaries and improve its usability. However, this forces QUERYX to rely on the accuracy of decompilers. Compared to lifting to IRs [13, 14, 27, 35], decompilation is more complex and thus error-prone. For instance, QUERYX failed to discover a bug candidate in §8.2 due to incorrect type inference of IDA Hex-Rays (i.e., `idx[arr]` instead of `arr[idx]`). We believe that such issues can be mitigated by advanced type inferences [33] and semantics-preserving decompilation [54, 74]. Notably, we employed public symbols for Windows binaries and symbols in the automotive binary to help decompilation. IDA Hex-Rays successfully decompiled 99.85% (11,184 / 11,199) of the functions in three Windows kernel drivers of §7.

Depending on Queries. Similar to other extensible static checking tools, QUERYX heavily depends on the quality of its query. If a query is incomplete or too general, QUERYX may result in many false alarms. In contrast, if analysts carefully guide the analysis, QUERYX can produce accurate results. For example, the bug candidates found by QUERYX in §7.3 were all true positives due to its high-quality query including vtable resolution and inter-procedural analysis. We thus made query writing as intuitive as possible: QUERYX queries are based on decompiled code, which is more analyst-friendly than IRs, and employs a fearless symbolic analysis, which reduces the manual efforts for scalability.

False Positives. QUERYX has false positives because of its under-constrained symbolic execution, incomplete queries, and inaccurate decompilation. Notably, two authors required three days (24 hours) to manually analyze every case in §7. This was not too difficult because the false positives exhibited distinct patterns as previously mentioned. Analysts can further reduce false positives by encoding additional rules into queries. For example, to reduce false positives in §7.1, we first manually figure out some of the functions that only the administrator can trigger. We then make queries dismiss them and other functions that can be only triggered from them. Additionally, advanced decompilation can reduce false positives due to improper decompilation.

Multi-decompiler Support. Although the current version of QUERYX only supports IDA Hex-Rays as its back-end decompiler, we believe that QUERYX can also support multiple decompilers seamlessly, thanks to DNR in QUERYX. Particularly, after lifting, QUERYX only uses DNR in the remaining procedures for analysis. Therefore, QUERYX can support other decompilers such as Ghidra [50], Binary Ninja [65], or angr [56] if we implement lifting decompiled code, which is obtained from other decompilers, into DNR.

10. Related work

Decompilation. For many binary analysts, decompilation is critical because it supports high-level data such as variable types and control-flow. Thus, many decompilers have been proposed, such as IDA Hex-Rays, Ghidra [50], Binary Ninja [65], Retdec [37], and angr [55, 56]. Moreover, Phoenix [54] performs semantics-preserving structural analysis and iterative control-flow structuring to recover high-level information of binary code. In addition, DREAM [73] and DREAM^{++} [75] improve the readability of decompiled code by several semantics-preserving code transformations. With the rise of big data, many data- and learning-driven decompilers [20, 32, 34, 53] also have been proposed. The current QUERYX is based on IDA Hex-Rays but can employ other decompilers as we discussed in §9.

Static Analysis for Finding Specific Bugs. Static analysis is one of the widely used techniques for bug finding [16, 21, 38, 39, 59, 78]. To analyze complex software like kernels or browsers, there have been static analyzers that focus on specific types of bugs [2, 40, 51, 60, 67, 69, 77, 79–81]. For example, KINT [68] detects more than 100 integer overflow bugs in the kernel. CGSan [26] finds use-after-compacting-gc bugs in browsers. Wang *et al.* [66] and Deadline [72] perform pattern matching and tailored symbolic checking to detect double-fetch bugs in the kernel, respectively. Moreover, Brown *et al.* [6] discover various bugs in browsers by pattern matching with μcheck [5]. We believe that most of these patterns can be encoded as queries of QUERYX.

Extensible Static Checking. Extensible static checking tools have been widely used for writing application-specific rules for finding bugs [1, 12, 19, 24, 36, 41, 58, 70, 71]. For instance, Clang [36] supports a framework to implement static analysis on top of its compiler. CodeQL [22] successfully finds numerous bugs in real-world software using its extensible query system. Recently, Sys [7] proposes a two-step analysis to scale symbolic analysis in large software such as browsers but requires source code. And joern [76] supports decompiled code analysis but is not precise due to the lack of precise parsing and symbolic analysis.

Binary Symbolic Execution. Many binary analysis tools support symbolic execution because of its usefulness [4, 11, 23, 57]. To manage the complexity of binary code and support extensibility, these tools mostly rely on IRs. For instance, angr [55], BAP [9], BINSEC [15], and B2R2 [31] employ PyVEX [49], BAP Intermediate Language (BIL), Dynamic Bit-vector Automata (DBA), and LowUIR, respectively. Unlike these tools, QUERYX's query is based on decompiled code, which is more high-level and analyst-friendly than binary IRs.

Scalable Symbolic Execution. Symbolic execution [10, 10, 23, 25, 56] is one of the most successful techniques for automatic bug discovery, but it suffers from scalability issues due to path explosions. There has been much research regarding path explosions. One common way to mitigate path explosions is to skip uninteresting paths. For example, Bergan *et al.* [3] start symbolic execution from any program states while approximating states by context-specific data-flow analysis. And UC-KLEE [52] begins symbolic execution from function entries with undefined states. Chopper [64] skips paths that analysts mark as uninteresting while WOODPECKER [12] and Firmalice [55] filter out uninteresting parts of programs by relationships to events and program dependency graphs, respectively. Inspired by the above research, QUERYX employs under-constrained symbolic execution and CFG reduction to skip uninteresting paths.

11. Conclusion

In this paper, we have presented QUERYX, the first extensible static checking tool based on decompiled code for fearless symbolic analysis. QUERYX suggests a new IR, DNR, for decompiled code and performs binary-aware analysis to be accurate. For scalability, QUERYX reduces CFG by callbacks and ordering dependencies between them. We evaluated QUERYX on the Windows kernel, Windows system service, and an automotive binary with four queries. As a result, we found 15 previously unknown bugs including 10 CVEs and received $180,000 from Microsoft as a bug bounty. We also experimentally demonstrated that CFG reduction can significantly improve the scalability of QUERYX and QUERYX outperformed existing extensible static checking tools in terms of bug findings and query writing.

Acknowledgment

We thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01202, Regional strategic industry convergence security core talent training business).

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2002.

[2] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[3] T. Bergan, D. Grossman, and L. Ceze. Symbolic execution of multithreaded programs from arbitrary program contexts. In *Proceedings of the 25th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, Oct. 2014.

[4] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[5] F. Brown, A. Nötzli, and D. Engler. How to build static checking systems using orders of magnitude less code. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[6] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in javascript bindings. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[7] F. Brown, D. Stefan, and D. Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug. 2020.

[8] F. Brown, D. Stefan, and D. Engler. Sys: A static/symbolic tool for finding good bugs in good (browser) code. https://github.com/PLSysSec/sys, 2020.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, July 2011.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[11] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[12] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[13] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, June 2019.

[14] S. Dasgupta, S. Dinesh, D. Venkatesh, V. S. Adve, and C. W. Fletcher. Scalable validation of binary lifters. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2020.

[15] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M.-L. Potet, and J.-Y. Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016.

[16] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at facebook. *Communications of the ACM*, 62(8):62–70, 2019.

[17] A. Dutcher. How to get Vex-IR for an entire function? https://github.com/angr/pyvex/issues/97#issuecomment-361747627, 2018.

[18] J. Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/, 2013.

[19] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Oct. 2000.

[20] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. Coda: An end-to-end neural program decompiler. In *Proceedings of the 2019 Advances in Neural Information Processing Systems*, Dec. 2019.

[21] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[22] Github Inc. CodeQL. https://codeql.github.com/, 2006.

[23] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2008.

[24] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

[25] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[26] H. Han, A. Wesie, and B. Pak. Precise and scalable detection of use-after-compacting-garbage-collection bugs. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[27] N. Hasabnis, R. Qiao, and R. Sekar. Checking correctness of code generator architecture specifications. In *Proceedings of the 2015 International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, Feb. 2015.

[28] Hex-Rays SA. IDA Pro - Hex Rays. https://hex-rays.com/ida-pro/, 2022.

[29] A. Hidayat. Esprima. http://esprima.org/, 2011.

[30] R. Jhala and R. Majumdar. Path slicing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[31] M. Jung, S. Kim, H. Han, J. Choi, and S. K. Cha. B2R2: Building an efficient front-end for binary analysis. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2019.

[32] D. S. Katz, J. Ruchti, and E. Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.

[33] O. Katz, R. El-Yaniv, and E. Yahav. Estimating types in binaries using predictive modeling. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, St. Peterburg, FL, Jan. 2016.

[34] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[35] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In *Proceedings of the 32rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017.

[36] T. Kremenek. Finding software bugs with the clang static analyzer. https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf, 2008.

[37] J. Křoustek, P. Matula, and P. Zemek. Retdec: An open-source machine-code decompiler, 2017.

[38] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security*

Symposium (Security), Washington, DC, Aug. 2001.

[39] J. Lawall and G. Muller. Coccinelle: 10 years of automated evolution in the linux kernel. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[40] K. Lu, A. Pakki, and Q. Wu. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[41] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[42] A. Mantovani, L. Compagna, Y. Shoshitaishvili, and D. Balzarotti. The convergence of source code and binary vulnerability discovery–a case study. In *Proceedings of the 17th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Nagasaki, Japan, May–June 2022.

[43] Microsoft. Windows kernel elevation of privilege vulnerability, CVE-2021-31979. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-31979, 2021.

[44] Microsoft. LOBYTE macro. https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms632658(v=vs.85), 2022.

[45] MITRE. CVE-2019-1477. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1477, 2019.

[46] MITRE. CVE-2020-1081. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-1081, 2020.

[47] MITRE. CVE-2021-31979. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31979, 2021.

[48] L. D. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, Mar.–Apr. 2008.

[49] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[50] NSA. Ghidra. https://ghidra-sre.org/, 2019.

[51] A. Pakki and K. Lu. Exaggerated error handling hurts! an in-depth study and context-aware detection. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual, Nov. 2020.

[52] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[53] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[54] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using Semantics-Preserving structural analysis and iterative Control-Flow structuring. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[55] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[56] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[57] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International conference on information systems security*, 2008.

[58] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 1994.

[59] Synopsys Inc. Coverity scan. https://scan.coverity.com/, 2006.

[60] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. Autoises: Automatically inferring security specification and detecting violations. In *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, July–Aug. 2008.

[61] The angr team. Optimization considerations. https://github.com/angr/angr-doc/blob/master/docs/speed.md, 2022.

[62] The Clang team. Clang static analyzer. https://clang-analyzer.llvm.org, 2007.

[63] The cppcheck team. Cppcheck. https://cppcheck.sourceforge.io, 2007.

[64] D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenbug, Sweden, May–June 2018.

[65] Vector 35. Binary Ninja. https://binary.ninja/, 2016.

[66] P. Wang, J. Krinke, K. Lu, and G. Li. How double- fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[67] W. Wang, K. Lu, and P.-C. Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[68] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[69] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu. Understanding and detecting disordered error handling with precise function pairing. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[70] Y. Xie and A. Aiken. Saturn: A SAT-based tool for bug detection. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV)*, Scotland, UK, July 2005.

[71] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, Long Beatch, CA, Jan. 2005.

[72] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[73] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[74] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[75] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[76] F. Yamaguchi. Joern – the bug hunter's workbench. https://joern.io/, 2014.

[77] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery.

In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[78] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[79] H. Yan, Y. Sui, S. Chen, and J. Xue. Machine-learning-guided typestate analysis for static use-after-free detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2017.

[80] H. Yan, Y. Sui, S. Chen, and J. Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenbug, Sweden, May–June 2018.

[81] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. APISan: Sanitizing API Usages through Semantic Cross-checking. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

```
function isChecked(state, src) {
  return hasCheck(state, src, '/')
         && hasCheck(state, src, '\\');
}

// Goal: Find a path without proper validation
function symRule(node) {
  if (isCall(node,"wcsrchr") && isSeparator(node.args[1])) {
    setCallback(node, function (node, state){
      var str = state.getValue(node);
      var sep = state.getValue(node.args[1]);
      addPath(state, str);
      addCheck(state, str, sep);
    })
  }

  if (isCall(node,"StringCchCatW")) {
    var src = node.args[2];
    if (isString(src) && hasSeparator(src)) {
      setCallback (node, function (node, state) {
        var dst = state.getValue(node.args[0]);
        addPath(state, dst);
      });
    } else {
      setCallback (node, function (node, state) {
        var dst = state.getValue(node.args[0]);
        var src = state.getValue(node.args[2]);
        if (isPath(state, src)) addPath(state, dst);
        if (isPath(state, dst)) addPath(state, src);
        if (isPath(state, dst) && !isChecked(state, src))
          print("Path traversal detected");
      });
    }
  }
}
```

(a) A query for finding path traversal bugs.

```
unsigned __int16 *__fastcall UniDrvUI::PConcatFilename(...)
{
  ...
  tmp = wcsrchr(fileName, '\\');
  if ( tmp )
    fileName = tmp + 1;
  ...
  StringCchCatW(path, pathSize, fileName);
  ...
}
```

(b) CVE-2020-17042, one of the path traversal bugs found by QUERYX.

Figure 8: A query for finding path traversal bugs and one of the bugs found by QUERYX.

Appendix A. Structure of DNR

MetaData	μ	=	`ExprInfo` * `AstInfo`
Expression	*exp*	=	`Num` (*value*, *size*, μ)
		|	`Undefined` (*size*, μ)
		|	`ProgramAddr` (*offset*, μ)
		|	`Arg` (*idx*, *size*, μ)
		|	`Var` (*name*, *size*, μ)
		|	`TempVar` (*name*, *size*, μ)
		|	`UnOp` ($\diamondsuit_u$, *exp*, μ)
		|	`BinOp` ($\diamondsuit_b$, *exp*, *exp*, μ)
		|	`RelOp` ($\diamondsuit_r$, *exp*, *exp*, μ)
		|	`ITE` (*exp*, *exp*, *exp*, μ)
		|	`Load` (*exp*, *size*, μ)
		|	`Extract` (*exp*, *pos*, *size*, μ)
		|	`Cast` ($\diamondsuit_c$, *size*, *exp*, μ)
		|	`Call` (*name*, *exp* [], *size*, μ)
		|	`IndCall` (*exp*, *exp* [], *size*, μ)
Statement	*stmt*	=	`Define` (*name*, *exp*)
		|	`Alloc` (*name*, *size*)
		|	`Store` (*exp*, *exp*)
		|	`Jmp` (*blockID*)
		|	`CondJmp` (*exp*, *blockID*, *blockID*)
		|	`Return` (*exp*)
Basic Block	*block*	=	*stmt* []
Function	*func*	=	*name* * *addr* * *args* * *block* []
Program Data	*progData*	=	*section* []
Program	*prog*	=	*func* [] * *progData*

Appendix B. Information of Target Binaries in §7

Binary	Version	Size	Time
ntfs.sys	10.0.19041.1081	2.8MB	15m
fastfat.sys	10.0.19041.1348	416KB	1m
win32kfull.sys	10.0.19041.1526	3.7MB	15m
PrintConfig.dll	10.0.18362.1198	3.4MB	10m
Automotive	REDACTED	6.9MB	56m

TABLE 7: Version, size, and decompilation time of each target binary in §7

Appendix C. Finding Path Traversal Bugs

We wrote a query for discovering logic bugs rather than memory corruptions to demonstrate that QUERYX can handle a variety of bugs. We created a query for the path traversal bug, which is one of the most famous logic bugs. We were inspired by previous path traversal bugs in Windows system services for printers: CVE-2019-1477 [45] and CVE-2020-1081 [46]. We applied the query to PrintConfig.dll, which is a Windows system service for printers.

Query. Figure 8a shows the query for finding path traversal bugs. This bug is caused by the absence of checks for whether a user-controllable path can contain directory separators (‘\’ and ‘/’ in Windows) when constructing a full path. We first identified which value is a path by simple heuristics instead of relying on a tricky list of path-related APIs. Particularly, we considered a value as a path if it had a check or was a string with directory separators (Line 8 – 15 and Line 19 – 24). We then recursively identified a path if it is used to construct a full path through concatenation (Line 28 – 29). Finally, we concluded the code as a path traversal bug if a path is not checked with directory separators before concatenation (Line 30 – 31).

Result. QUERYX took 1.5 min to analyze PrintConfig.dll using the query in Figure 8a and found 8 bug candidates. After manual analysis, we confirmed 2 bugs and 6 false positives. One of the found bugs, which was assigned to CVE-2020-17042, is shown in Figure 8b. The vulnerable function only checks whether `fileName` has ‘\’ and constructs a path with it. Therefore, we can bypass the check by using ‘/’ instead of ‘\’, and this leads to a path traversal bug.

False Positives. As we described above, the query treats values that contain directory separators as file paths. However, Windows registry paths also have a separator, ‘\’, which is identical to one of the directory separators. This causes QUERYX to misjudge Windows registry paths as file paths, thereby resulting in false positives. For example, QUERYX reported that a path traversal vulnerability exists in `AssembleRegistrySubkey` because it confused Windows registry paths with file paths.

Bug	QueryX			angr			joern	CodeQL
	Time	Total	Detected	Time	Total	Detected	Detected	Detected
CVE-2017-1000249	5s	1	✓	3m	1	✓	✓	✓
CVE-2013-6462	30s	1	✓	6s	2	✓	✓	✓
BUG-2012	-	-	×	-	-	×	×	×
CVE-2017-6298	4s	1	✓	24s	1	✓	✓	✓
CVE-2018-11360	13h	13	✓	> 24h	303	✓	×	×
CVE-2017-17760	2.5m	19	✓	1.8h	30	✓	✓	×
CVE-2019-19334	40s	1	✓	1.1h	1	✓	✓	✓
CVE-2019-1010315	20m	4	✓	13m	4	×	✓	✓
BUG-2010	3s	1	✓	12m	2	✓	×	×
BUG-2018	2.6h	2	✓	> 24h	18	✓	✓	×

TABLE 8: Results of QueryX, angr, joern, and CodeQL when targeting 10 bugs selected by Mantovani *et al.* [42].

Appendix D.
Heap OOB access bug in SQLite Sys missed but QueryX found

In §8.2, there were some bugs that QueryX found but Sys could not, and Figure 9 depicts one of them. Note that we successfully wrote a proof-of-concept that triggered this heap OOB bug. In Line 8–11, `fts3ContentColumns` function calculates the total length of the column names, `nStr`. Then, it allocates `nStr + 8 * nCol` bytes to `azCol` and copies the column names into `azCol`. Due to the lack of an integer overflow check for the allocation size, this function results in a heap OOB. The Sys query missed this bug because it failed to cover all LLVM IR instructions in SQLite even though this query was 6× longer than that of QueryX. This highlights that query writing in QueryX is simpler and more intuitive than that in Sys.

Appendix E.
Comparison against angr, joern, and CodeQL on the dataset from Mantovani *et al.*

To compare QueryX against existing extensible static checking tools on different targets from §7, we evaluated them on 10 bugs from Mantovani *et al.* [42]. Note that Mantovani *et al.* selected 10 real-world bugs to evaluate the feasibility of applying source code analysis tools to analyze decompiled code. Although Mantovani's work also includes other generic static analysis tools (e.g., CPPCheck [63] or Clang Static analyzer [62]), we only compared QueryX against joern and CodeQL. This is because, as stated in that paper, comparing these tools with generic ones is rather unfair because they rely on custom queries. As a result, joern and CodeQL outperformed other generic tools in detecting vulnerabilities during their evaluation.

For comparison, we carefully wrote queries of QueryX and angr by referring to bug descriptions and existing queries of joern and CodeQL in Mantovani *et al.*. This is because extensible static checking tools are highly dependent on their query qualities. In particular, we attempted to make their queries nearly identical to those of joern and CodeQL, with the exception that symbolic analysis is more natural. For example, we symbolically verify that the copy size argument of `memcpy` is greater than the buffer size for detecting overflow unlike in the case of joern and CodeQL, which check the existence of any size check (e.g., a conditional statement with a less than operator).

We ran QueryX and angr with queries for 10 bugs in the same configuration as §7. For joern and CodeQL, we borrowed the results from Mantovani *et al.*. Notably, these tools employ distinct analysis objects to detect vulnerabilities (Table 1). QueryX, CodeQL, and joern use decompiled code, whereas angr uses VEX IR from binary code. Additionally, CodeQL assumes that the source code can be compiled because it is originally designed for source code analysis. As a result, Mantovani *et al.* had to manually edit the decompiled code to make it compilable and use it for CodeQL, which drastically limited the usability of CodeQL for binary code.

Table 8 depicts that QueryX and angr succeed to find 9 and 8 bugs while joern and CodeQL found 7 and 5 bugs, respectively. This means that QueryX can support diverse types of bugs and targets. QueryX outperforms joern and CodeQL because these tools rely on several assumptions from source code, which do not hold in decompiled code, as noted §5 in Mantovani *et al.*. For example, joern and CodeQL missed BUG-2010 because their taint tracking was not binary-aware.

Compared to QueryX, angr failed to discover one bug and took 2.2× more time in total. In particular, angr missed CVE-2019-1010315 due to different memory modeling from that of QueryX, as we described in §8.1. Furthermore, angr also returned more false positives than QueryX when finding CVE-2018-11360; CVE-2018-11360 is a null-byte overflow in Wireshark due to the lack of boundary checks when writing a null value into an array. Similar to the queries for joern and CodeQL from Mantovani *et al.*, QueryX can determine which memory access is for an array from decompiled code, but angr cannot. Therefore, angr needs to analyze every null-byte write, thereby resulting in many false positives. Moreover, angr is quite slow in analyzing extremely large binaries; for example, angr fails to finish analyzing the vulnerabilities in Wireshark within 24 hours (i.e., CVE-2018-11360 and BUG-2018), which is a 77MB binary with 2 million lines of code. During this experiment, angr frequently invoked garbage collection to reclaim memory because angr suffered from high memory usage. This is a well-known issue of angr [61]. Notably, none of these tools can detect BUG-2012 because they cannot locate a memory allocation function in the target, which was invoked by an indirect call. We confirmed that QueryX can discover BUG-2012 if we encode this indirect call pattern in the query.

```
1  __int64 __fastcall fts3ContentColumns(...) {
2    int n; // [rsp+24h] [rbp-7Ch]
3    int nCol; // [rsp+48h] [rbp-58h]
4    int nStr; // [rsp+4Ch] [rbp-54h]
5    ...
6    nStr = 0;
7    nCol = chrome_sqlite3_column_count(pStmt);
8    for ( i = 0; (int)i < nCol; ++i ) {
9      zCol = (char *)chrome_sqlite3_column_name(pStmt, i);
10     nStr += strlen(zCol) + 1;
11   }
12   azCol = chrome_sqlite3_malloc((unsigned int)(nStr + 8 * nCol));
13   if ( azCol ) {
14     p = (char *)(8LL * nCol + azCol);
15     for ( j = 0; (int)j < nCol; ++j ) {
16       zCol_ = (char *)chrome_sqlite3_column_name(pStmt, j);
17       n = strlen(zCol_) + 1;
18       memcpy(p, zCol_, n);
19       ...
20 }
```

Figure 9: A heap out-of-bound access bug in SQLite QueryX found but Sys did not.