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

(TheCurrently postdoc in Georgia Tech)
Static Analysis

• One of the most popular techniques for automatic bug finding

Scalability vs Accuracy
Extensible Static Checking Tools

- Static analysis + Domain knowledge (Query)

Source Code
- C / C++
- LLVM IR
- Java
- ...

Domain knowledge

Query

CodeQL
joern
Sys (Sec ’20)
...

Bug Candidates
# Extensible Static Checking Tools

- Static analysis + Domain knowledge (Query)

<table>
<thead>
<tr>
<th>Input</th>
<th>Analysis objects</th>
<th>Syntactic</th>
<th>Data-flow</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeQL [22]</td>
<td>Source</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>joern [76]</td>
<td>Source</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Sys [7]</td>
<td>Source LLVM IR</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Binary ???**
Extensible Static **Binary** Checking Tools

- Query based on **binary IRs**

- **Binary IR**
  - Vex IR
  - BIL
  - ...

- **Query**

- **Domain knowledge**

- **angr**
  - BAP
  - ...

- **Bug Candidates**
Extensible Static **Binary** Checking Tools

- Query based on **binary IRs**

  - Domain knowledge
  - Binary IR
    - Vex IR
    - BIL
    - ...

  - angr
  - BAP

  - Give me decompiled code instead of binary IR

  - Bug Candidates
Inconsistency b/w Analyst & Query

• Analysts mostly work with **decompiled code** due to its high-level information such as high-level control flows and types.
• But, queries are based on **binary IRs** because binary IRs are more analyzer-friendly.

```c
__int64 __fastcall isPalindrome(const char *a1) {
    int i; // [rsp+18h] [rbp-8h]
    int v3; // [rsp+1Ch] [rbp-4h]

    v3 = strlen(a1);
    for ( i = 0; i < v3 / 2; ++i ) {
        if ( a1[i] != a1[v3 - i - 1] )
            return 0LL;
    }
    return 1LL;
}
```

```Assembly
00 | ------ IMark(0x40069a, 1, 0) ------
01 | t0 = GET:I64(rbp)
02 | t10 = GET:I64(rsp)
03 | t9 = Sub64(t10,0x0000000000000008)
04 | PUT(rsp) = t9
05 | STle(t9) = t0
06 | ------ IMark(0x40069b, 3, 0) ------
07 | PUT(rbp) = t9
08 | ------ IMark(0x40069e, 4, 0) ------
09 | t2 = Sub64(t9,0x0000000000000020) ; + 285 more instructions
```
# Extensible Static Binary Checking Tools

- Static analysis + Domain knowledge (Query)

<table>
<thead>
<tr>
<th>Input</th>
<th>Analysis objects</th>
<th>Syntactic</th>
<th>Data-flow</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeQL [22]</td>
<td></td>
<td></td>
<td></td>
<td>Cannot handle binaries</td>
</tr>
<tr>
<td>joern [76]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sys [7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>angr [56]</td>
<td>Binary</td>
<td>VEX IR</td>
<td></td>
<td>Consistency b/w Analyst &amp; Query</td>
</tr>
<tr>
<td>BAP [9]</td>
<td>Binary</td>
<td>BIL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Decompiled code
The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study (AsiaCCS ‘22)

- Feed decompiled code from Hex-Rays to CodeQL and joern.
- Compared to tools with the original source code, tools with decompiled code got less TP and more FP because...

```
int glob_x = 0x42;
struct Pos { int x; int y; };  
int main() {
    struct Pos pos;
    pos.x = glob_x;
    pos.y = 0x43;
    print_pos(&pos);
}
```

```
int main() {
    int v1;  // [rsp+0h] [rbp-20h]
    int v2;  // [rsp+4h] [rbp-1Ch]
    v1 = dword_201;
    v2 = 0x43;
    print_pos(&v1);
}
```

Original code

Decompiled code
The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study (AsiaCCS ‘22)

- Feed decompiled code from Hex-Rays to CodeQL and joern.
- Compared to tools with the original source code, tools with decompiled code got **less TP** and **more FP** because..

**Previous source code analysis tools are binary-unaware!**

```c
pos.y = 0x43;
print_pos(&pos);
}
```

```c
v2 = 0x43;
print_pos(&v1);
}
```
**Symbolic**

**QueryX: Query on Decompile Code**

- Static analysis + Domain knowledge (Query)

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>Analysis objects</th>
<th>Syntactic</th>
<th>Data-flow</th>
<th>Symbolic</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeQL [22]</td>
<td>Source</td>
<td>Source</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>joern [76]</td>
<td>Source</td>
<td>Source</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>Sys [7]</td>
<td>Source</td>
<td>LLVM IR</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>angr [56]</td>
<td>Binary</td>
<td>VEX IR</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>BAP [9]</td>
<td>Binary</td>
<td>BIL</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td><strong>QUERYX</strong></td>
<td>Binary</td>
<td>Decompile code</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Goals & Approaches of QueryX

1. Binary-aware analysis on decompiled code
   • Analysis based on our new IR, DNR

2. Analyst-friendly symbolic query
   • Symbolic query based on decompiled code and callbacks
   • JavaScript-like query

3. Scalable analysis with analyst-friendly symbolic query
   • Under-constrained symbolic execution
   • CFG reduction based on callbacks and their dependencies

=> Check our paper
QueryX Architecture

1. **Binary**
   - **Decompiler**
     - **Query Interpreter**
       - **QueryX**
         - **Lifter**
           - **Analyzer**
             - Syntactic Matching
             - Data-flow Analysis
             - Symbolic Analysis

2. **Query**
   - **Result**

- Decompiled Code
- Analysis Rule
- Analysis Result
- DNR
Goals & Approaches of QueryX

1. Binary-aware analysis on decompiled code
   • Analysis based on our new IR, DNR

2. Analyst-friendly symbolic query
   • Symbolic query based on decompiled code and callbacks
   • JavaScript-like query

3. Scalable analysis with analyst-friendly symbolic query
   • Under-constrained symbolic execution
   • CFG reduction based on callbacks and their dependencies
Decompiler-Neutral Representation (DNR)

```c
int main() {
    int v1; // [rsp+0h] [rbp-20h]
    int v2; // [rsp+4h] [rbp-20h]

    // ISSUE1: Binary-embedded data
    v1 = dword_2010A4;
    v2 = 0x43;

    // ISSUE2: Binary-dependent code
    print_pos(&v1);
}
```

Resolve by adding Program Data to DNR

Missing global variable information (e.g., initial value, RWX permission)
Decompiler-Neutral Representation (DNR)

```c
int main() {
    int v1; // [rsp+0h] [rbp-20h]
    int v2; // [rsp+4h] [rbp-1Ch]

    // ISSUE1: Binary-encoding
    v1 = dword_2010A4;
    v2 = 0x43;

    // ISSUE2: Binary-dependence
    print_pos(&v1);
}
```

Resolve by considering memory layout while lifting

real binary (w/ memory layout):
- X: 0x42
- Y: 0x43

w/o memory layout:
- X: 0x42
- Y: 0xff // undefined value
Decompiler-Neutral Representation (DNR)

```c
int main() {
    int v1; // [rsp+0h] [rbp-20h]
    int v2; // [rsp+4h] [rbp-1Ch]

    // ISSUE1: Binary-embedded data
    v1 = dword_2010A4;
    v2 = 0x43;

    // ISSUE2: Binary-dependent code
    print_pos(&v1);
}
```

```
stack = Alloc(0x8)
// v1: @(stack + 0x0)
// v2: @(stack + 0x4)

Store(stack,
    Load(ProgAddr(0x2010a4), 4),
    4)
Store(stack + 0x4, 0x43, 4)

Call(print_pos, [|stack|])
```
Decompiler-Neutral Representation (DNR)

So, should we write queries based on DNR?

We can write queried based on decompiled code because DNR contains which decompiled code is lifted to the corresponding DNR.
Goals & Approaches of QueryX

1. Binary-aware analysis on decompiled code
   • Analysis based on our new IR, DNR

2. Analyst-friendly symbolic query
   • Symbolic query based on decompiled code and callbacks
   • JavaScript-like query

3. Scalable analysis with analyst-friendly symbolic query
   • Under-constrained symbolic execution
   • CFG reduction based on callbacks and their dependencies
QueryX’s Symbolic Query Example (1)

• A simple query for finding simple heap overflow

```c
char src[0x100];
...
buf = malloc(size);
if (cond)
    memcpy(buf, src, 0x100);
```

The copy size of memcpy can be greater than the size of dst.
QueryX’s Symbolic Query Example (1)

- A simple query for finding simple heap overflow

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

function symRule (node) {
    if (isCall(node, "malloc"))
        setCallback(node, collectAlloc, "alloc");

    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}
```

Perform symexec from the entry of `func` based on `symRule`
A simple query for finding simple heap overflow

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

function symRule (node) {
    if (isCall(node, "malloc"))
        setCallback(node, collectAlloc, "alloc");

    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}
```

Recursively traverse AST nodes in `func` and call `symRule`.
QueryX’s Symbolic Query Example (1)

- A simple query for finding simple heap overflow

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

def symRule (node) {
    if (isCall(node, "malloc"))
        setCallback(node, collectAlloc);
    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}
```

Register `collectAlloc` as callbacks of `malloc` call nodes

Register `checkMemcpy` as callbacks of `memcpy` call nodes
QueryX’s Symbolic Query Example (1)

• A simple query for finding simple heap overflow

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

function symRule (node) {
    if (isCall(node, "malloc"))
        setCallback(node, collectAlloc);

    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}
```

Save allocated addresses and their sizes to the current symstate.

Check whether the copy size can be greater than the size of dst.
QueryX’s Symbolic Query Example (1)

- A simple query for finding simple heap overflow

```javascript
function collectAlloc (node, state) {
  // node: AST node of malloc call
  // state: Current symbolic state
  var addr = state.getValue(node);
  var size = state.getValue(node.args[0]);
  if (state.allocs == undefined) state.allocs = [];
  // Store allocated address and its size
  state.allocs.push({addr: addr, size: size});
}
```
QueryX’s Symbolic Query Example (1)

• A simple query for finding simple heap overflow

```javascript
function checkMemcpy (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")
        }
    }
}
```

Find the size of dst

Check whether the copy size can be greater than the size of dst under the current path constraints.
QueryX’s Symbolic Query Example (1)

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

function symRule (node) {
    if (isCall(node, "malloc"))
        setCallback(node, collectAlloc, "alloc");

    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}

function collectAlloc (node, state) {
    // node: AST node of malloc call
    // state: Current symbolic state
    var addr = state.getValue(node);
    var size = state.getValue(node.args[0]);
    if (state.allocs == undefined) state.allocs = [];
    // Store allocated address and its size
    state.allocs.push({addr: addr, size: size});
}

function checkMemcpy (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");
        }
    }
}
```
QueryX’s Symbolic Query Example (2)

• A query for finding heap overflow due to integer overflow in Windows kernel

```c
__int64 __fastcall RtlpCreateServerAcl(...) {
    unsigned short AclSize = 8;
    ...
    if ((_WORD)AceCount) {
        for (i = 0; i < AceCount; i++) {
            AclSize += Aces[i]->AceSize;
        }
    }
    buffer = ExAllocatePoolWithTag(PagedPool, AclSize, 'cAes');
    ...
    AclOffset = 0;
    for (i = 0; i < AceCount; i++) {
        ...
        memcpy(buffer + AclOffset, Aces[i], Aces[i]->AceSize);
        AclOffset += AclSize;
    }
}
```

```c
function symRule (node) {
    if (isCall(node, "ExAllocatePoolWithTag")
        && node.args[1].type.size == 2
        && !isConstant(node.args[1]))
        setCallback(node, collectAlloc, "alloc");

    if (isCall(node, "memcpy"))
        setCallback(node, checkMemcpy, "copy");
}
```

CVE-2021-31979 (Heap overflow due to integer overflow)
QueryX’s Symbolic Query Example (2)

CVE-2021-41378, one of heap overflow bugs QueryX found

```c
NTSTATUS __fastcall NtfsSetDispositionInfo(...)
{
    ...
    length = volumeName->Length + dirName->Length
             + fileName->Length;
    ...
}
```

<table>
<thead>
<tr>
<th>Idx</th>
<th>CVE</th>
<th>Program</th>
<th>Function</th>
<th>Bug Type</th>
<th>Bounty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CVE-2021-41370</td>
<td>ntfs.sys</td>
<td>NtfsSetShortNameInfo</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>2</td>
<td>CVE-2021-41378</td>
<td>ntfs.sys</td>
<td>NtfsSetDispositionInfo</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>3</td>
<td>CVE-2021-43229</td>
<td>ntfs.sys</td>
<td>TxfAllocateAndStoreNameForTxfLogging</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>4</td>
<td>CVE-2021-43230</td>
<td>ntfs.sys</td>
<td>TxfAllocateFullPathForChangeNotify</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>5</td>
<td>CVE-2021-43231</td>
<td>ntfs.sys</td>
<td>NtfsRenameToPrivateDir</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>6</td>
<td>CVE-2021-41367</td>
<td>ntfs.sys</td>
<td>TxfOpenFileProcessing</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>7</td>
<td>CVE-2022-23293</td>
<td>fastfat.sys</td>
<td>FatSetFullNameInFcb</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
</tbody>
</table>

...
angr vs QueryX in the Example

Binary IRs do not have such information (e.g., type, constant value..)

angr: 267 LoC, QueryX: 33 LoC
Evaluation (1-day)

- Comparison on the Dataset of Mantovani et al. (“The convergence of source code and binary vulnerability discovery—a case study”, AsiaCCS ’22)

<table>
<thead>
<tr>
<th>Bug</th>
<th>Total</th>
<th>Detected</th>
<th>QueryX</th>
<th>joern</th>
<th>CodeQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2017-1000249</td>
<td>5s</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2013-6462</td>
<td>30s</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BUG-2012</td>
<td>-</td>
<td>-</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CVE-2017-6298</td>
<td>4s</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2018-11360</td>
<td>13h</td>
<td>13</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2017-17760</td>
<td>2.5m</td>
<td>19</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2019-19334</td>
<td>40s</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CVE-2019-1010315</td>
<td>20m</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BUG-2010</td>
<td>3s</td>
<td>1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BUG-2018</td>
<td>2.6h</td>
<td>2</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Evaluation (1-day)

- Comparison on the Dataset of Mantovani et al. ("The convergence of source code and binary vulnerability discovery—a case study", AsiaCCS '22)

Lack of high-level info (e.g., type)

Binary-unaware

<table>
<thead>
<tr>
<th>Bug</th>
<th>QUERYX</th>
<th>angr</th>
<th>joern</th>
<th>CodeQL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Total</td>
<td>Detected</td>
<td>Time</td>
</tr>
<tr>
<td>CVE-2017-1000249</td>
<td>5s</td>
<td>1</td>
<td>✔</td>
<td>3m</td>
</tr>
<tr>
<td>CVE-2013-6462</td>
<td>30s</td>
<td>1</td>
<td>✔</td>
<td>6s</td>
</tr>
<tr>
<td>BUG-2012</td>
<td>-</td>
<td>-</td>
<td>×</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2017-6298</td>
<td>4s</td>
<td>1</td>
<td>✔</td>
<td>24s</td>
</tr>
<tr>
<td>CVE-2018-11360</td>
<td>13h</td>
<td>13</td>
<td>✔</td>
<td>&gt; 24h</td>
</tr>
<tr>
<td>CVE-2017-17760</td>
<td>2.5m</td>
<td>19</td>
<td>✔</td>
<td>1.8h</td>
</tr>
<tr>
<td>CVE-2019-19334</td>
<td>40s</td>
<td>1</td>
<td>✔</td>
<td>1.1h</td>
</tr>
<tr>
<td>CVE-2019-1010315</td>
<td>20m</td>
<td>4</td>
<td>✔</td>
<td>13m</td>
</tr>
<tr>
<td>BUG-2010</td>
<td>3s</td>
<td>1</td>
<td>✔</td>
<td>12m</td>
</tr>
<tr>
<td>BUG-2018</td>
<td>2.6h</td>
<td>2</td>
<td>✔</td>
<td>&gt; 24h</td>
</tr>
</tbody>
</table>
Evaluation (0-day)

- QueryX found 15 previous unknown vulnerabilities including **10 CVEs** and earned **$180,000** from MS bug bounty program.

<table>
<thead>
<tr>
<th>Idx</th>
<th>CVE</th>
<th>Program</th>
<th>Function</th>
<th>Bug Type</th>
<th>Bounty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CVE-2021-41370</td>
<td>ntfs.sys</td>
<td>NtfsSetShortNameInfo</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>2</td>
<td>CVE-2021-41378</td>
<td>ntfs.sys</td>
<td>NtfsSetDispositionInfo</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>3</td>
<td>CVE-2021-43229</td>
<td>ntfs.sys</td>
<td>TxfAllocateAndStoreNameForTxfLogging</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>4</td>
<td>CVE-2021-43230</td>
<td>ntfs.sys</td>
<td>TxfAllocateFullPathForChangeNotify</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>5</td>
<td>CVE-2021-43231</td>
<td>ntfs.sys</td>
<td>NtfsRenameToPrivateDir</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>6</td>
<td>CVE-2021-41367</td>
<td>ntfs.sys</td>
<td>TxfOpenFileProcessing</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>7</td>
<td>CVE-2022-23293</td>
<td>fastfat.sys</td>
<td>FatSetFullNameInFcb</td>
<td>Heap overflow</td>
<td>$20,000</td>
</tr>
<tr>
<td>8</td>
<td>CVE-2022-30162</td>
<td>win32kfull.sys</td>
<td>NtUserSetClassLongPtr</td>
<td>Kernel Address Disclosure</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>CVE-2020-17041</td>
<td>PrintConfig.dll</td>
<td>CopyFileFromPrinterData</td>
<td>Path Traversal</td>
<td>$20,000</td>
</tr>
<tr>
<td>10</td>
<td>CVE-2020-17042</td>
<td>PrintConfig.dll</td>
<td>UniDrvUtil::PConcatFilename</td>
<td>Path Traversal</td>
<td>$20,000</td>
</tr>
<tr>
<td>11−15</td>
<td></td>
<td>Automotive</td>
<td>REDACTED</td>
<td>Out-of-bound Access</td>
<td>-</td>
</tr>
</tbody>
</table>
More in the paper

• Scalable analysis with analyst-friendly symbolic query

• 4 kinds queries for finding 0-days

• More detail comparison against other tools

• Other evaluations and details
Thank you