

APISan: Sanitizing API Usages through Semantic Cross-checking

Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang,
Taesoo Kim, Mayur Naik

Georgia Institute of Technology

APIs in today's software are plentiful yet complex

- Example: OpenSSL
 - **3841** APIs in [v1.0.2h]
 - 3718 in [v1.0.1t] -> 3841 in [v1.0.2h] (**+123** APIs)
 - OpenSSH uses **158** APIs of OpenSSL



mozilla
Firefox[®]

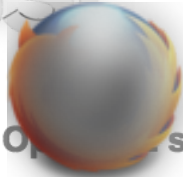


Complex APIs result in programmers' mistakes

- Problems in documentation
 - Incomplete: e.g., low details in hostname verification
 - Long: e.g., 43K lines in OpenSSL documentation
 - Lack: e.g., internal APIs
- Lack of automatic tool support
 - e.g., missing formal specification and precise semantics

Problem: API misuse can cause security problems

OpenSSl



mozilla

Firefox



#2008-016 multiple OpenSSl signature verification API misuse

Description:

Several functions ins

This bug allows a ma

The flaw may be ex
validation.

CVE-ID

CVE-2014-4113

[Learn more at National Vulnerability Database \(NVD\)](#)

• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

win32k.sys in the kernel-mode drivers in Microsoft Windows Server 2003 SP2, Windows Vista SP2, Win Windows 8, Windows 8.1, Windows Server 2012 Gold and R2, and Windows RT Gold and 8.1 allows loc exploited in the wild in October 2014, aka "Win32k.sys Elevation of Privilege Vulnerability."

References

Note: [References](#) are provided for the convenience of the reader to help distinguish between vulnerabilities. The list

→ Privilege Escalation

Today's practices to help programmers

- Formal method
 - Problem: lack of specification
- Model checking
 - Problem: manual, lack of semantic context
- Symbolic execution
 - Problem : failed to scale for large software

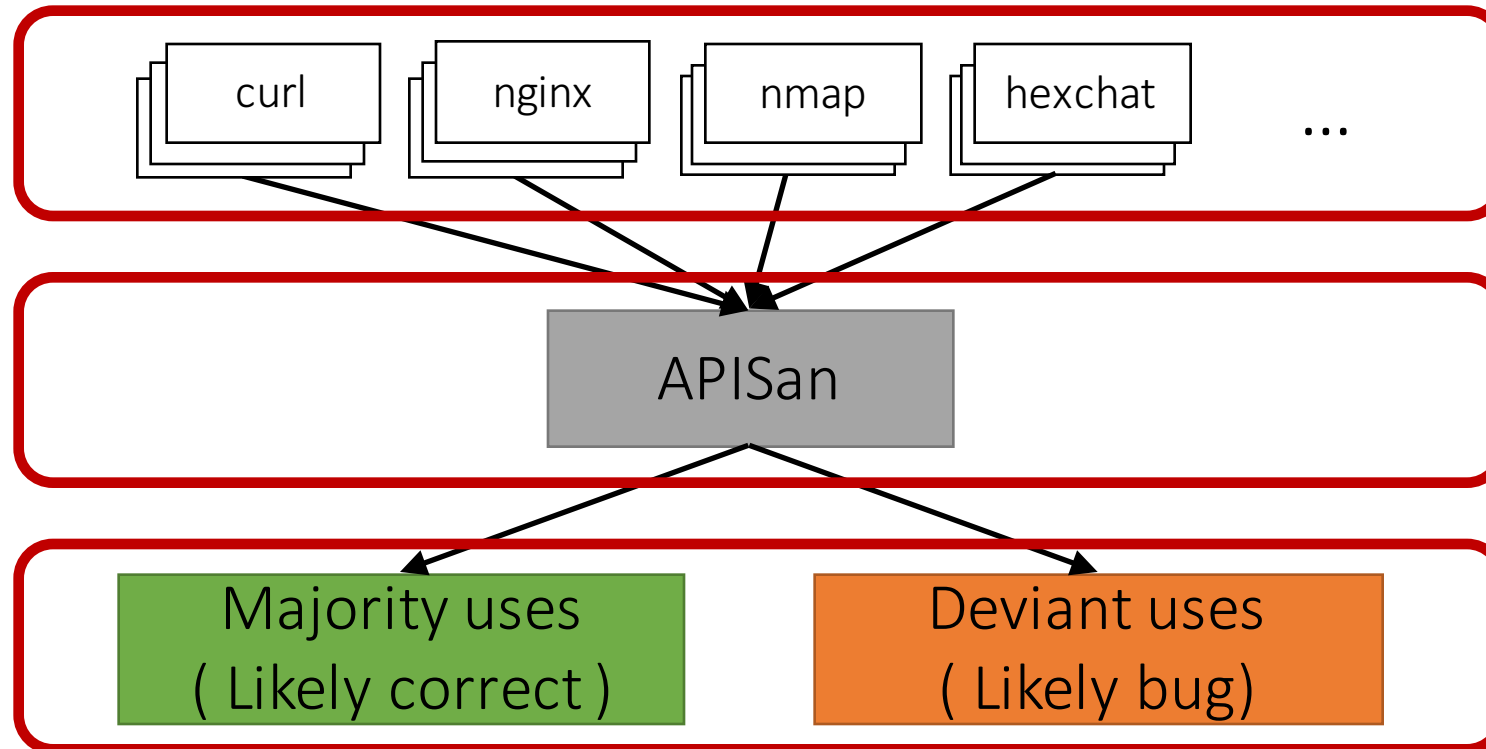
Promising approach: finding bugs by using existing code

- “Bugs as deviant behavior” [OSDI01]
 - Syntactic template: e.g., check NULL on malloc()

Research goal: can we apply this method to *any* kind of software *without manual efforts*?

Our idea: comparing API usages in various implementation

- Example: finding OpenSSL API misuses



Our approach is very promising

- Effective in finding API misuses
 - 76 new bugs
- Scale to large, complex software
 - Linux kernel, OpenSSL, PHP, Python, etc.
 - Debian packages

Technical Challenges

- API uses are too different from impl. to impl.
- Subtle semantics of the correct API uses
- Large, complex code using APIs

Example: OpenSSL API uses

- `SSL_get_verify_result()`
 - Get result of peer certificate verification
 - **no peer certificate → always returns `X509_V_OK`**

```
if (SSL_get_verify_result() == X509_V_OK) { ... }  
    && SSL_get_peer_certificate() != NULL ) { ... }
```

Example: a correct implementation using OpenSSL API

Semantically same with correct usage

```
if (SSL_get_verify_result() == X509_V_OK  
&& SSL_get_peer_certificate() != NULL ) { ... }
```

Example: providing various implementations using OpenSSL

Correct

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);
```

Correct

```
if (SSL_get_verify_result(conn) != X509_V_OK)  
return NGX_OK;  
cert = SSL_get_peer_certificate(conn);
```

Can we distinguish between *correct* implementations and *buggy* implementations?

```
if (cert == NULL)  
return 0;  
if (SSL_get_verify_result(ssl) != X509_V_OK) {...}
```

nmap

```
case X509_V_OK:  
cert = SSL_get_peer_certificate(ssl);  
// if (cert) is missed
```

hexchat

Challenge 1: API usages are different from each other

Correct

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) { ... }
```

curl

Correct

```
if (SSL_get_verify_result(conn) != X509_V_OK)  
    return NGX_OK;  
cert = SSL_get_peer_certificate(conn);  
if (cert) { ... }
```

nginx

Correct

```
cert = SSL_get_peer_certificate(ssl);  
if (cert == NULL)  
    return 0;  
if (SSL_get_verify_result(ssl) != X509_V_OK) {...}
```

nmap

Incorrect

```
err = SSL_get_verify_result(ssl);  
switch(err) {  
    case X509_V_OK:  
        cert = SSL_get_peer_certificate(ssl);  
        // if (cert) is missed
```

hexchat

Challenge 2: subtle semantics of the correct API usages

Correct

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) { ... }
```

curl

Correct

```
if (SSL_get_verify_result(conn) != X509_V_OK)  
    return NGX_OK;  
cert = SSL_get_peer_certificate(conn);  
if (cert) { ... }
```

nginx

Correct

```
cert = SSL_get_peer_certificate(ssl);  
if (cert == NULL)  
    return 0;  
if (SSL_get_verify_result(ssl) != X509_V_OK) {...}
```

nmap

Incorrect

```
err = SSL_get_verify_result(ssl);  
switch(err) {  
    case X509_V_OK:  
        cert = SSL_get_peer_certificate(ssl);  
        // if (cert) is missed
```

hexchat

Challenge3 : Large, complex code using APIs

- On average, more than 100K LoC
 - curl : 110K LoC
 - nginx : 127K LoC
 - nmap: 169K LoC
 - hexchat: 61K LoC
- Linux : > 1M LoC

Challenge3 : Large, complex code using APIs

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) { ... }
```

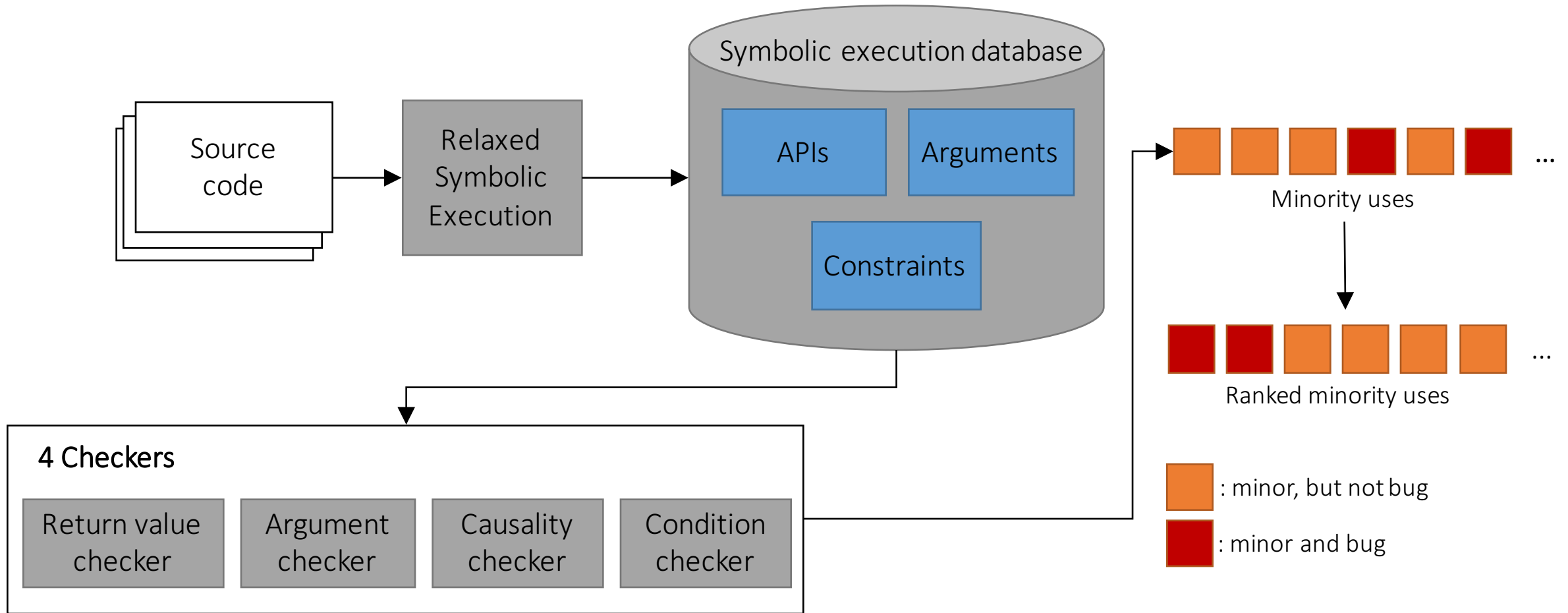
curl (simplified)



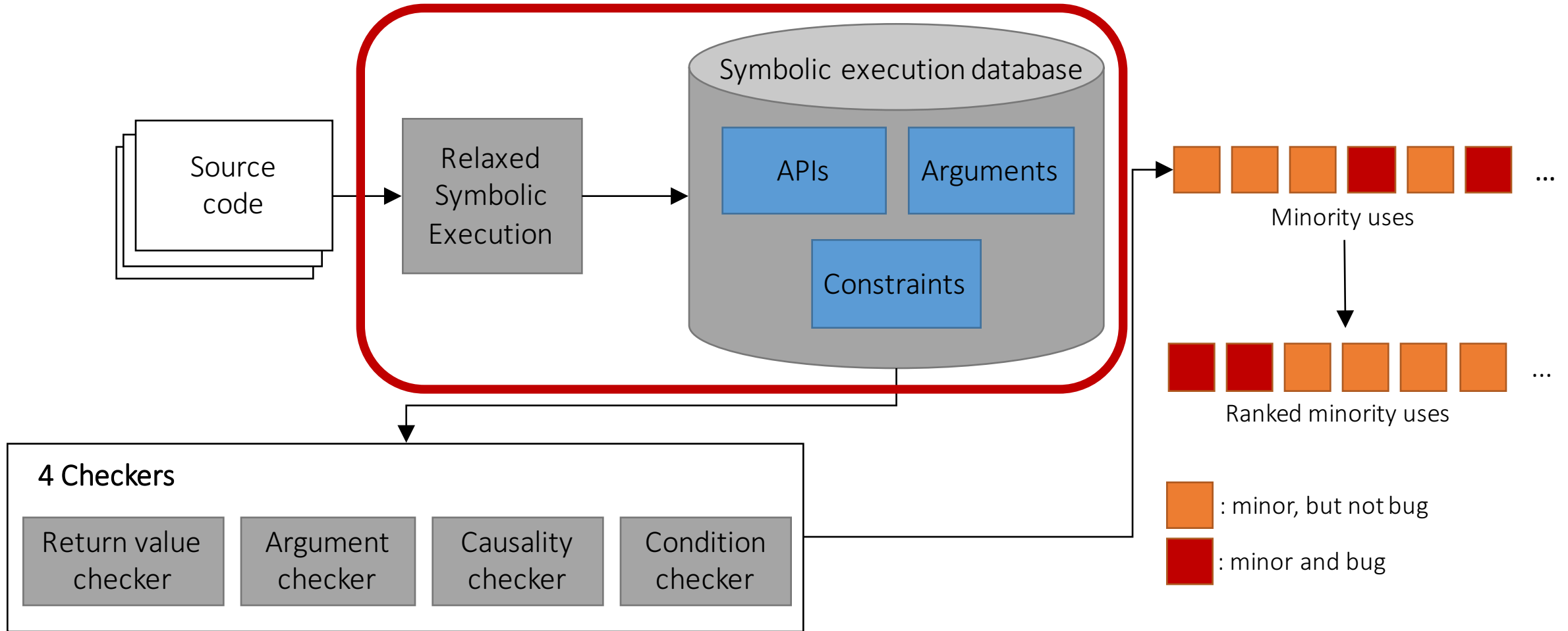
```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
...  
len = BIO_get_mem_data(mem, (char **) &ptr);  
infof(data, " start date: %.*s\n", len, ptr);  
rc = BIO_reset(mem);  
...  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) { ... }
```

curl

Overview of APISan



Overview of APISan



Symbolic execution can be relaxed in finding API contexts

- Symbolic execution is not scalable
 - Path explosion
 - SMT is expensive, naturally NP-complete
- Methods to relax symbolic execution
 - Limiting inter-procedural analysis
 - Removing back edges
 - Range-based

Method 1: Limiting inter-procedural analysis

- How APIs are used O
- How APIs are implemented X

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err != X509_V_OK) { ... }
```

Method 2: Removing back edges

- API contexts can be captured within loops
 - e.g., malloc() and free() are matched inside a loop

```
for(...){  
    cert = SSL_get_peer_certificate(handle);  
    if (!cert) {...}  
    err = SSL_get_verify_result(handle);  
    if (err != X509_V_OK) { ... }  
}
```

Method 3: Range-based

- Most of arguments & return values are integer

```
cert != NULL  $\wedge$  err == X509_V_OK
```



```
cert = {[-MAX, -1] , [1, MAX]}  
err = {[X509_V_OK, X509_V_OK]}
```



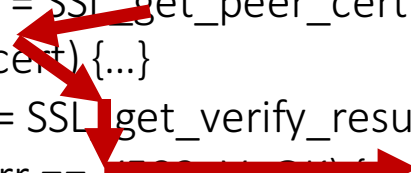
- Clang uses range-based symbolic execution

Building per-path symbolic abstractions

- Path-sensitive, context-sensitive
- Record symbolic abstractions
 - API calls
 - Symbolic expression of arguments
 - Constraints

Examples: Building per-path symbolic abstractions from source code

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) {...}
```



Source code

Call	SSL_get_peer_certificate(handle)
Constraint	SSL_get_peer_certificate(handle) = {[-MAX, -1], [1, MAX]}
Call	SSL_get_verify_result(handle)
Constraint	SSL_get_verify_result(handle) = {[X509_V_OK, X509_V_OK]}

Symbolic abstractions

Examples: Building per-path symbolic abstractions from source code

```
cert = SSL_get_peer_certificate(handle);  
if (!cert) {...}  
err = SSL_get_verify_result(handle);  
if (err == X509_V_OK) {...}
```

Source code

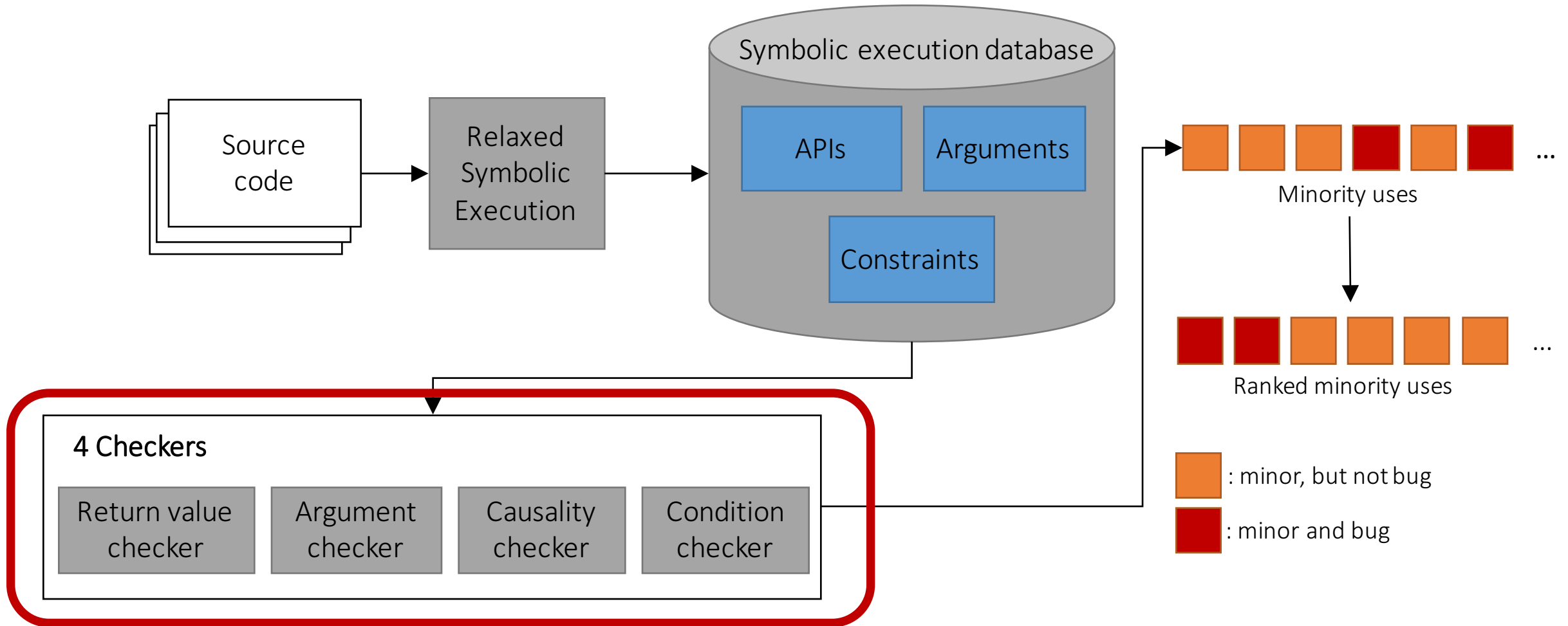
Symbolic
Abstractions #1

Symbolic
Abstractions #2

Symbolic
Abstractions #3

....

Overview of APISan



Four semantic contexts have security implications

- Orthogonal, essential, security-related contexts
 - Return value
 - Arguments
 - Causality
 - Condition

Context 1: Return value

- Return computation result or execution status
- NULL dereference
- Privilege escalation
 - e.g, Windows, CVE-2014-4113

```
ptr = malloc(size)  
if (!ptr){ ... }
```

Context 2: Arguments

- Inputs for calling APIs and their relationship
- Format string bug
- Memory corruption

```
printf(buf);
```

```
ptr = malloc(size1);  
memcpy(ptr, src, size2);
```

Context 3: Causality

- Causal relationship between APIs
- Deadlock
- Memory leak

```
lock();  
unlock();
```

```
malloc();  
free();
```

Context 4: Condition

- Implicit pre- and post condition for calling APIs
- MITM

```
if (SSL_get_verify_result() == X509_V_OK &&  
    SSL_get_peer_certificate() != NULL)
```

Extract contexts from symbolic abstractions

- Symbolic abstractions contains **{APIs, Arguments, Constraints}**
- Return value ← Constraints
- Arguments ← Arguments
- Causality ← APIs
- Condition ← Constraints + APIs

Example: extract condition contexts from symbolic abstractions

Call	SSL_get_peer_certificate(handle)
Constraint	SSL_get_peer_certificate(handle) = {[-MAX, -1], [1, MAX]}
Call	SSL_get_verify_result(handle)
Constraint	SSL_get_verify_result(handle) = {[X509_V_OK, X509_V_OK]}

curl

Any constraint or call	Line numbers when event is called
Event	Line
SSL_get_verify_result = {[X509_V_OK, X509_V_OK]}	{curl}
Constraint	Line
SSL_get_peer_certificate = {[-MAX, -1], [1, MAX]}	{curl}
...

Example: extract condition contexts from symbolic abstractions

Call	SSL_get_verify_result(conn)
Constraint	SSL_get_verify_result(handle) == {[X509_V_OK, X509_V_OK]}
Call	SSL_get_peer_certificate(conn)
Constraint	SSL_get_peer_certificate(conn) != [-MAX, -1], [1, MAX]}

nginx

Event	Line
SSL_get_verify_result = {[X509_V_OK, X509_V_OK]}	{curl, nginx }
Constraint	Line
SSL_get_peer_certificate = [-MAX, -1], [1, MAX]}	{curl, nginx }
...

Example: extract condition contexts from symbolic abstractions

Call	SSL_get_peer_certificate(ssl)
Constraint	SSL_get_peer_certificate(ssl) = {[-MAX, -1], [1, MAX]}
Call	SSL_get_verify_result(ssl)
Constraint	SSL_get_verify_result(ssl) = {[X509_V_OK, X509_V_OK]}

nmap

Event	Line
SSL_get_verify_result = {[X509_V_OK, X509_V_OK]}	{curl, nginx, nmap }
Constraint	Line
SSL_get_peer_certificate = {[-MAX, -1], [1, MAX]}	{curl, nginx, nmap }
...

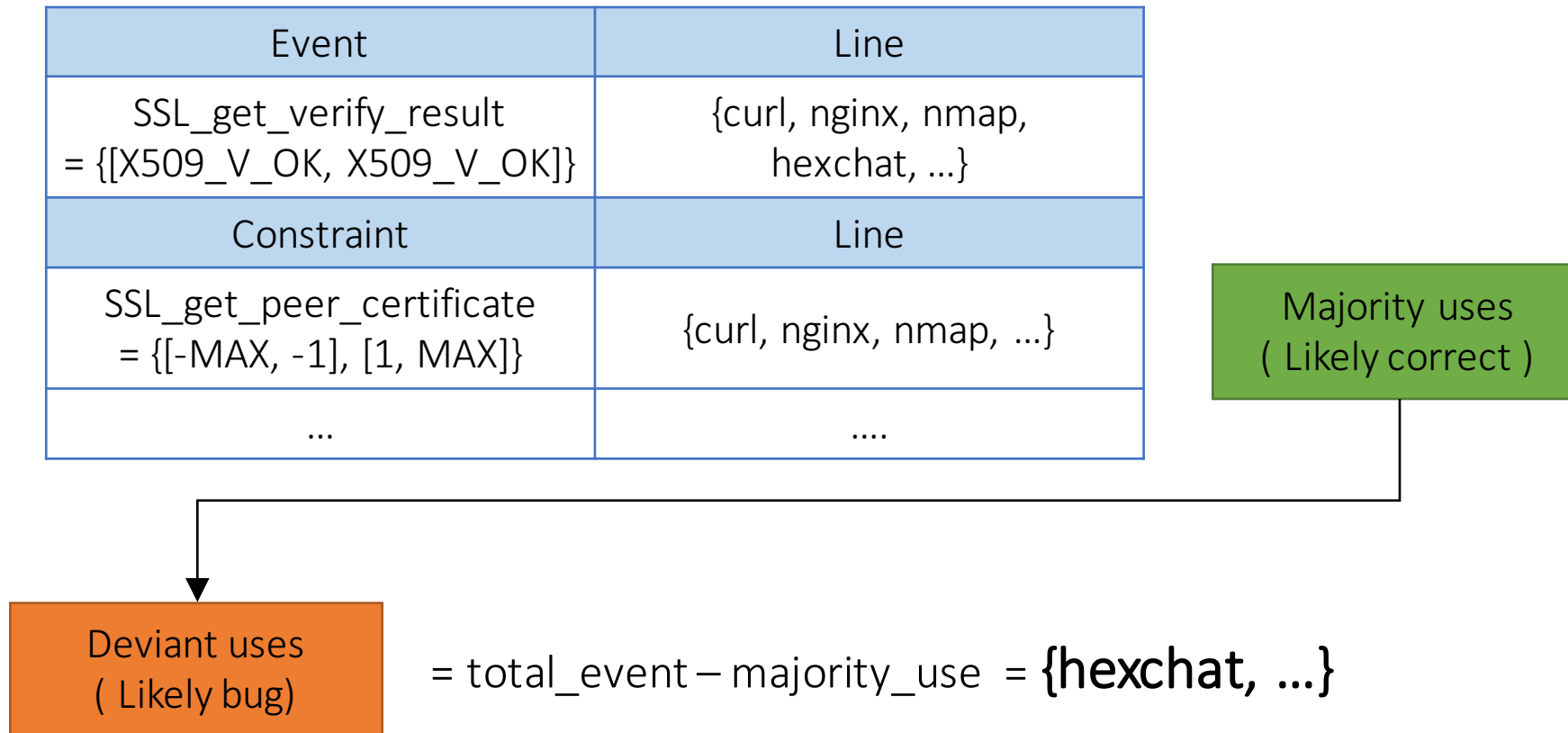
Example: extract condition contexts from symbolic abstractions

Call	SSL_get_verify_result(ssl)
Constraint	SSL_get_verify_result(ssl) = {[X509_V_OK, X509_V_OK]}
Call	SSL_get_peer_certificate(ssl)

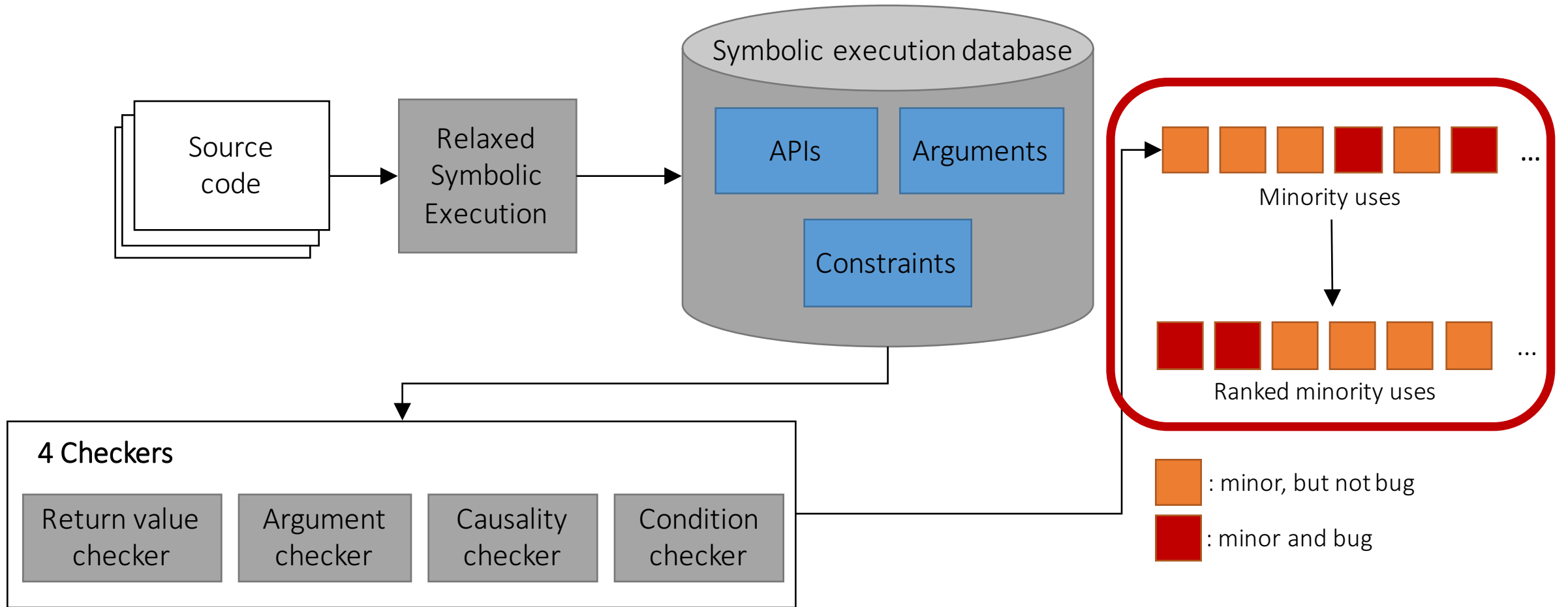
hexchat

Event	Line
SSL_get_verify_result = {[X509_V_OK, X509_V_OK]}	{curl, nginx, nmap, hexchat }
Constraint	Line
SSL_get_peer_certificate = {[-MAX, -1], [1, MAX]}	{curl, nginx, nmap}
...

Example: find majority & minority usages from contexts



Overview of APISan



False positives can be happened in majority analysis

- Lack of inter-procedural analysis
 - e.g., check a return value of malloc() inside a function
- Correlation \neq Causation
 - e.g., fprintf() is used for printing debug messages when open() is failed
- Correct minor uses
 - e.g., strcmp() == 0, strcmp() > 0

Ranking can mitigate false positives

- More majority pattern repeated, more bug-likely
 - e.g., 999 majority, 1 minority > 10 majority, 1 minority
- General information
 - e.g., most of allocation functions have “alloc” in their names and are required to check their return values
- Domain specific knowledge
 - e.g., SSL APIs start with a string “SSL”

Our approach is formalized as a general framework

$$\begin{aligned}
 \text{SymbolicContexts}(f) &= \{ (t, i, C) \mid t \in \mathbb{D} \wedge i \in [1..|t|] \wedge t[i] \equiv \mathbf{call} f(*) \wedge C = \text{CONTEXTS}(t, i) \} \\
 \text{Frequency}(f, c) &= \{ (t, i) \mid \exists C: c \in C \wedge (t, i, C) \in \text{SymbolicContexts}(f) \} \\
 \text{Majority}(f) &= \{ c \mid |\text{Frequency}(f, c)| / |\text{SymbolicContexts}(f)| \geq \theta \} \\
 \text{BugReports}(f) &= \{ (t, i, C) \mid (t, i, C) \in \text{SymbolicContexts}(f) \wedge C \cap \text{Majority}(f) = \emptyset \} \\
 \text{BugReportScore}(f) &= 1 - |\text{BugReports}(f)| / |\text{SymbolicContexts}(f)| + \text{HINT}(f)
 \end{aligned}$$

$$\begin{aligned}
 \text{returnValueContexts} &= \lambda(t, i). \{ \bar{r} \mid \exists j: t[j] \equiv \mathbf{assume}(e, \bar{r}) \wedge \langle \mathbf{ret}, i \rangle \in \text{retvars}(e) \} \\
 \text{argRelationContexts} &= \lambda(t, i). \{ (u, v) \mid t[i] \equiv \mathbf{call} *(\bar{e}) \wedge \text{argvars}(\bar{e}[u], t) \cap \text{argvars}(\bar{e}[v], t) \neq \emptyset \} \\
 \text{causalityContexts}\langle \bar{r} \rangle &= \lambda(t, i). \{ g \mid \exists j: t[j] \equiv \mathbf{assume}(e, \bar{r}) \wedge \langle \mathbf{ret}, i \rangle \in \text{retvars}(e) \wedge \exists k > j: t[k] \equiv \mathbf{call} g(*) \} \\
 \text{conditionContexts}\langle \bar{r} \rangle &= \lambda(t, i). \{ (g, \bar{r}') \mid \exists j: t[j] \equiv \mathbf{assume}(e, \bar{r}) \wedge \langle \mathbf{ret}, i \rangle \in \text{retvars}(e) \wedge \exists k > j: t[k] \equiv \mathbf{call} g(*) \wedge \\
 &\quad \exists l: t[l] \equiv \mathbf{assume}(e', \bar{r}') \wedge \langle \mathbf{ret}, k \rangle \in \text{retvars}(e') \} \\
 \text{defaultHint} &= \lambda f. 0 \quad \text{nullDerefHint} = \lambda f. \text{if } (f\text{'s name contains } \mathit{alloc}) \text{ then } 0.3 \text{ else } 0
 \end{aligned}$$

Implementation of APISan

- 9K LoC in total
 - Symbolic database generation : 6K LoC of C/C++ (Clang 3.6)
 - APISan library : 2K LoC of Python
- Checkers : 1K LoC of Python
 - Return value checker : 131 LoC
 - Argument checker : 251 LoC
 - ...

Evaluation questions

- How effective is APISan in finding new bugs?
- How easy to use and easy to extend?
- How effective is APISan's ranking system?

APISan is effective in finding bugs

- Found 76 new bugs in large, complex software
 - Linux kernel, OpenSSL, PHP, Python, and Debian packages
- Security implication
 - e.g., CVE-2016-5636: Python zipimporter heap overflow (Code execution in Google App Engine)

APISan is easy to use without any manual annotation

- To generate symbolic context database

```
$ apisan make # use existing build command
```

- Run a checker

```
$ apisan --checker=cpair # cpair : causality checker
```

- Run a checker (inter-application)

```
$ apisan --checker=cpair --db=app1, app2
```

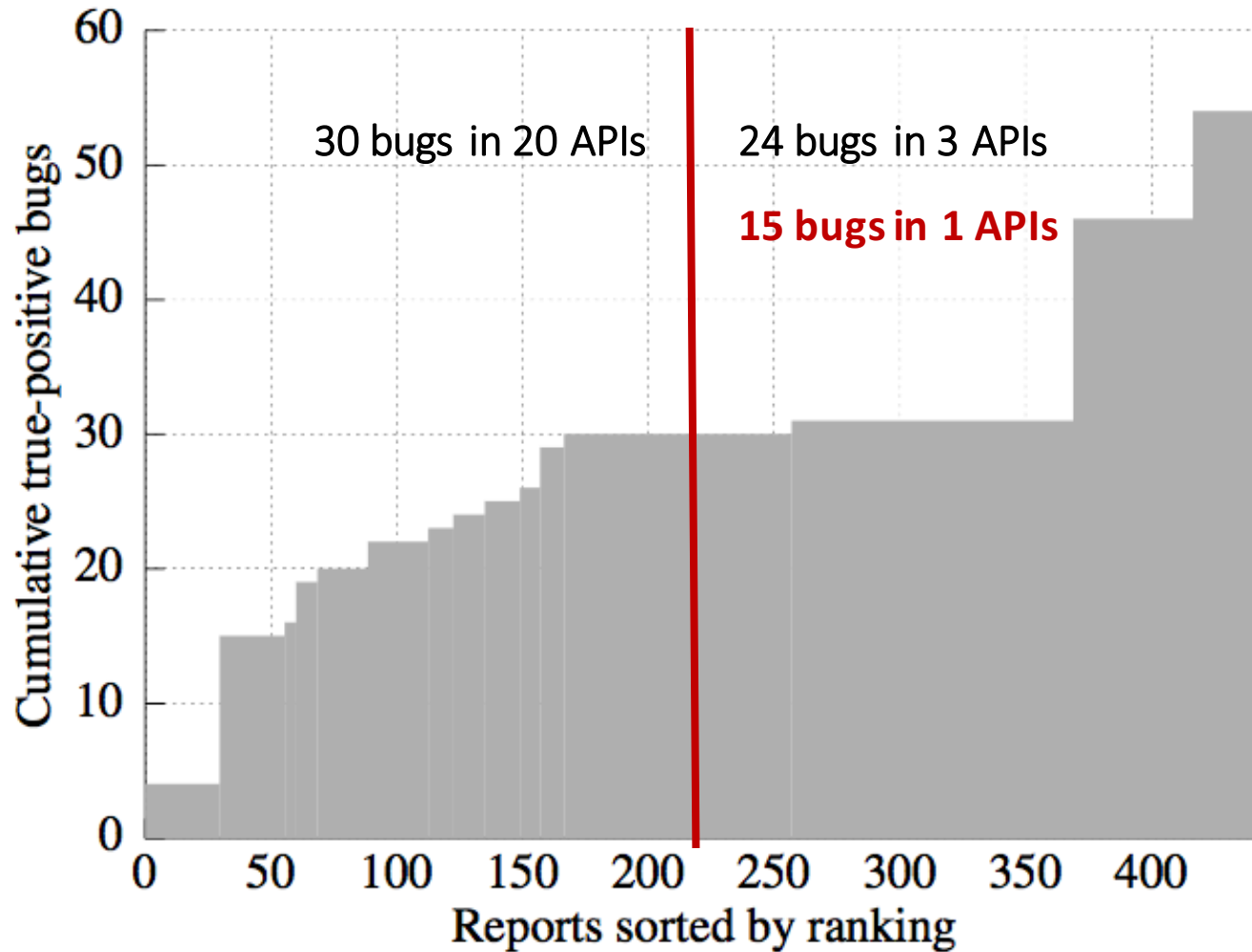
APISan is easy to extend

- e.g., Integer overflow check
- Integer overflow sensitive APIs
 - Have security implications when integer overflow happens
 - e.g., memory allocation functions
- Integer overflow ← Arguments + Constraints
 - If arguments contains binary operators
 - check integer overflow within given constraints

Check integer overflow with APISan

- Collect all integer overflows
- Ranking strategy
 - More integer overflow prevented by constraints
 - APIs are likely integer overflow sensitive
 - Incorrect constraints > Missing constraints
 - ; Missing constraints can be caused by limited analysis
- Found 6 integer overflows (167 LoC)

APISan's ranking system is effective



- Linux Kernel with Return Value Checker
- Total 2,776 reports
- Audited 445 reports
- Found 54 bugs

Limitation

- No soundness & No completeness
- High false positive rate : > 80%
- Too slow to frequently analyze
 - 32-core Xeon server with 256GB RAM
 - For Linux kernel,
Generating database : 8 hours
Each checker: 6 hours
- Not fully resolve path explosion
 - stopped in functions which have path explosion

Conclusion

- APISan: an automatic way for finding API misuse
 - Effective: Finding 76 new bugs
 - Scalable: Tested with Linux kernel, Debian packages, etc
- APISan ***WILL*** be released as open source
 - <https://github.com/sslab-gatech>

Thank you!

Questions?