Automated Counterexample Generation for Side Channels

By: Jameson DiPalma, Yuxiang Lin

Overview:

1. Background
2. Design/Implementation
3. Results/Discussion
4. Future Work

Background

Review: Constant Time (CT) Programming

- Programming principles designed to mitigate side channel attacks and protect secret information
- 2 types of violations: secret-dependent **branches** (line 3), and secret-dependent **memory accesses** (line 4)

```
1. int insecure_demo(int pub1, int* pub2, int secret) {
2.     if (pub1 > 0) {
3.         if (pub1 < 10 && secret > 0) {
4.             return pub2[secret];
5.         }
6.     }
7.     return -1;
8. }
```

Review: CtChecker[1]

- Programs can be deceptively difficult to write and verify as constant-time
- **CtChecker:** program analysis tool, finds CT-violations with few false negatives
- Reports line number of potential vulnerability;
  unhelpful for confirming and fixing leaks

```
library/bignum.c line  238 - if( Y->p[i] != 0 )
library/bignum.c line  365 - if( x & mask ) break;
library/bignum.c line  384 - if( X->p[i] != 0 )
library/bignum.c line 1045 - if( X->p[i - 1] != 0 )
library/bignum.c line 1049 - if( Y->p[j - 1] != 0 )
library/bignum.c line 1060 - if( X->p[i - 1] > Y->p[i - 1] ) return(  1 );
library/bignum.c line 1061 - if( X->p[i - 1] < Y->p[i - 1] ) return( -1 );
library/bignum.c line 1077 - if( X->p[i - 1] != 0 )
library/bignum.c line 1081 - if( Y->p[j - 1] != 0 )
library/bignum.c line 1095 - if( X->p[i - 1] > Y->p[i - 1] ) return(  X->s );
library/bignum.c line 1096 - if( X->p[i - 1] < Y->p[i - 1] ) return( -X->s );
library/bignum.c line 1145 - if( B->p[j - 1] != 0 )
library/bignum.c line 1162 - while( c != 0 )
library/bignum.c line 1225 - if( B->p[n - 1] != 0 )
library/bignum.c line 1245 - if( carry != 0 )
```

```
library/bignum.c line 1248 - for( ; n < X->n && X->p[n] == 0; n++ )
library/bignum.c line 1472 - while( n > 0 && A->p[n - 1] == 0 )
library/bignum.c line 1476 - if( b == 0 || n == 0 )
library/bignum.c line 1520 - if( 0 == d || u1 >= d )
library/bignum.c line 1520 - if( 0 == d || u1 >= d )
library/bignum.c line 1531 - if( quotient > ( (mbedtls_t_udbl) 1 << biL ) - 1 )
library/bignum.c line 1662 - if( X.p[i] >= Y.p[t] )
library/bignum.c line 1985 - if( mbedtls_mpi_cmp_int( N, 0 ) <= 0 || ( N->p[0] & 1 ) == 0 )
library/bignum.c line 2125 - if( ei == 0 && state == 0 )
library/bignum.c line 2128 - if( ei == 0 && state == 1 )
library/bignum.c line 2174 - if( ( wbits & ( one << wsize ) ) != 0 )
library/bignum.c line 2183 - if( neg && E->n != 0 && ( E->p[0] & 1 ) != 0 )
library/bignum.c line 1399 - *d += c; c = ( *d < c ); d++;
library/bignum.c line 1399 - *d += c; c = ( *d < c ); d++;
library/bignum.c line 1399 - *d += c; c = ( *d < c ); d++;
```

[1] Zhou, Quan, Sixuan Dang, and Danfeng Zhang. "CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming." (ECOOP 2024).

Review: Symbolic Execution

- Program inputs are treated as symbolic variables, not concrete values.
- The program is executed along all feasible paths, building conditions (path constraints) for each.
- A solver (like Z3) is used to check which paths are possible and find inputs that trigger them.

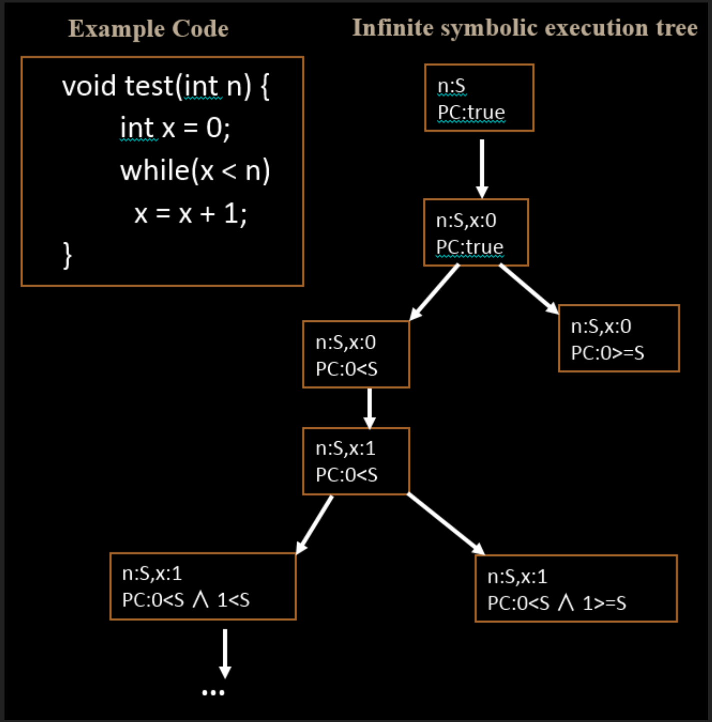


Image: Haj-Yahya, Jawad & Ben-asher, Yosi. (2018). Software Static Energy Modeling for Modern Processors. International Journal of Parallel Programming. 46.

Project Objective

- Use symbolic execution (SE) to find counterexamples of CT violations
- Focus on secret-dependent **branches**
- Counterexample: pair of inputs with different execution paths where only secret inputs differ
  - $(pub_1,\ \ldots,\ pub_n;\ sec_1^a,\ \ldots,\ sec_m^a)$
  - $(pub_1,\ \ldots,\ pub_n;\ sec_1^b,\ \ldots,\ sec_m^b)$
- Challenge: efficient modeling of side channels in SE

Design/Implementation

KLEE overview

- KLEE: popular open-source symbolic execution engine
- Explores feasible program paths by treating inputs as symbolic variables
- Operates on LLVM bitcode, an intermediate representation of the source level and machine-specific assembly
- Primary use in automatic test generation, producing concrete inputs that follow a particular path

Approach #1: Self-Composition

```
// Symbolically run code below using KLEE
void self_composition() {
    symbolic int pub, sec1, sec2;

    //augmented to record branch history
    target_func(pub, sec1, run=1);
    target_func(pub, sec2, run=2);

    assert(branch_histories_equal());
}
```

Advantages:

- Simple to implement
- Easily expandable to memory accesses
- Takes advantage of built-in KLEE search heuristics

Disadvantages:

- Requires exploring two full executions per counterexample; suffers from path explosion

Approach #2: Product Programs

- Check whether a branch is secret-dependent **when it's encountered (Does not need to wait until the end of a complete execution!)**
- For branch condition $\texttt{cond}(\texttt{pub}_1, \ldots, \texttt{pub}_n; \texttt{sec}_1, \ldots, \texttt{sec}_m)$, create two copies with secret variables renamed:
  - $\texttt{cond}^a(\texttt{pub}_1, \ldots, \texttt{pub}_n; \texttt{sec}_1{}^a, \ldots, \texttt{sec}_m{}^a)$
  - $\texttt{cond}^b(\texttt{pub}_1, \ldots, \texttt{pub}_n; \texttt{sec}_1{}^b, \ldots, \texttt{sec}_m{}^b)$
- Check if it is possible for the branch to diverge with the same public variables but different secret variables
  - $\texttt{solve}[\texttt{pub}_1, \ldots, \texttt{pub}_n; \texttt{sec}_1{}^a, \ldots, \texttt{sec}_m{}^a; \texttt{sec}_1{}^b, \ldots, \texttt{sec}_m{}^b]$
    $(\texttt{path\_cond} \wedge \texttt{cond}^a \wedge \neg\texttt{cond}^b)$

Approach #2: Product Programs

```
int insecure_demo(in                          cret) {
    if (pub1 > 0) {
        if (pub1 < 10 && secret > 0) {
            return pub2[secret];
}
```

Path Condition:

```
pub1 > 0
```

Solve:

```
pub1 > 0
&& (pub1 < 10 && secret^a > 0)
&& !(pub1 < 10 && secret^b > 0)
```

Counterexample:

```
pub1 == 1
&& secret^a == 1
&& secret^b == 0
```

Optimization: Concretization

- Issue: **solver timeout** on complex constraints (usually after adding a complex branch condition)
- Solution: concretize **all symbolic variables** to one set of fixed values when this happens
- Limitation: can only go down one path afterwards
    - Future improvements: concretize only some of the symbolic variables or concretize to multiple sets of values (concolic execution)

Optimization: Concretization

```
int insecure_demo(in                              cret) {
    if (pub1 > 0) {
        if (pub1 < 10 && secret > 0) {
            return pub2[secret];
        }
    }
    return -1;
}
```

Path Condition:
`pub1 > 0`

Execution continues
in one path

Concretization:
`pub1 == 1`
`&& secret == 1`

Results/Discussion

Counterexamples from CtChecker

- CPU: Intel(R) Xeon(R) Platinum 8352Y @ 2.20GHz
- Command: `klee --max-tests=64 --max-solver-time=10s  --libc=uclibc --posix-runtime`
- Running time: 7 hours

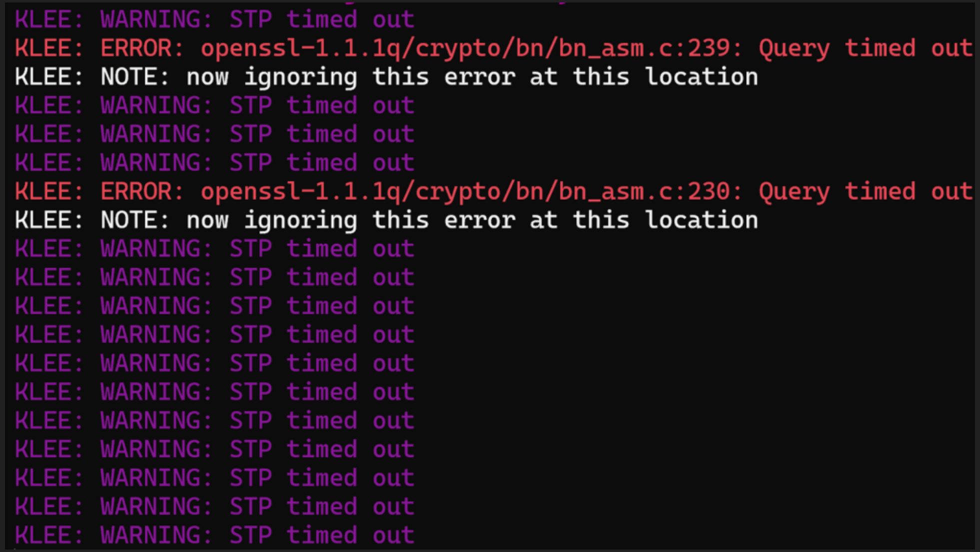
	PP-TP (Confirmed/Found)	PP-FP (Visited/Found)	SC
BearSSL 0.6	0/0	3/3	0
mbedTLS 3.2.1	3/4	17/26	2
Libgcrypt 1.10.1	0/6	8/26	N/A

- PP-TP: true positives confirmed by product program/found by CtChecker
- PP-FP: false positives visited by product program/found by CtChecker
- SC: number of divergent path pairs with concrete public and symbolic secret
- Fairly effective on mbedTLS (smaller), struggles with Libgcrypt (larger)

Limitations

Symbolic Execution:

- Path explosion
- Solver timeouts
- Difficulty modelling external calls



```
KLEE: WARNING: STP timed out
KLEE: ERROR: openssl-1.1.1q/crypto/bn/bn_asm.c:239: Query timed out
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: ERROR: openssl-1.1.1q/crypto/bn/bn_asm.c:230: Query timed out
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
KLEE: WARNING: STP timed out
```

Cryptographic Libraries:

- Individual bit logic, bignums
- Libgcrypt, openssl, etc.

  are much larger codebases

Future Work

Scalability Improvements

- Search heuristics: guide the exploration of paths towards target branches
- Solve constraints with different solvers simultaneously (e.g. STP + Z3)
- Program slicing: remove parts of the program irrelevant to target branches

```
1. a = 5;
2. b = a + a;
3. if (b > 0) {
4.   c = a;
5. }
6. d = b;
```

Slice criteria (6, b)

```
1. a = 5;
2. b = a + a;
3. d = b;
```

Extensions and Applications

-Generalize input generation to include memory-access-based time violations

-applications in quantifying/bounding leakage of side channel vulnerabilities

```
if (secret % 2 == 0) {
    return 1;
} else {
    return 0;
}
```

Leaks 1 bit of secret

```
int ret = 0;
for (int i = 0; i < 32; i++) {
    if ((secret >> i) & 1) {
        ret++;
    } else {
        continue;
    }
}
return ret;
```

Leaks all of secret

Summary

- CtChecker automatically finds security vulnerabilities, but without showing how they occur
- Augment KLEE, a symbolic execution engine, to produce concrete inputs that trigger secret-dependent control flow
- When encountering branch, see if both sides are simultaneously reachable while keeping public inputs fixed
- Effectively finds most true positives on smaller cryptographic libraries; issues with scalability
- To-do: parallelize constraint solvers, implement program slicing