Data flow analysis for Uranus applications

Chan Kwan Yin

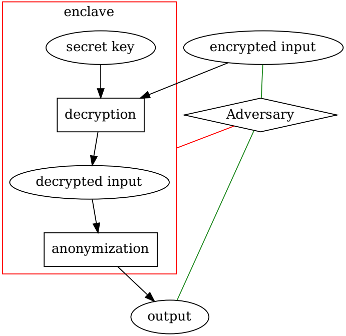
December 17, 2020

Outline

Background

SGX Enclaves

- Servers outsourced to third-party cloud providers
- Threat model: Adversaries with privileged access to OS, BIOS or hardware
- Enclave protects both code and memory from these adversaries



Uranus [2]

- OpenJDK fork that supports Intel SGX
- Methods marked as `@JECall` enters enclaves until return
- Methods marked as `@JOCall` exits enclaves until return
- Useful for integration with libraries like Hadoop and Spark
- Question: Where should `@JECall` and `@JOCall` be placed?
- Question: Is code in these libraries safe as enclave code?

The problem: Performanc/Security Tradeoff

- More code outside enclave:
  - Increased risk of leaking protected data
  - Some leaks may come from unexpected side channels
- More code into enclave:
  - Limited EPC (Enclave Page Cache)
    - Up to 100 MB of EPC
    - Out of memory $\implies$ extremely slow swap
    - JVM applications especially memory-greedy
  - Principle of Least Privilege
    - Vulnerabilities in enclave code bypass enclave protection
    - Vulnerabilities in code outside must only use specific entry points
    - Reduce attack surface

The solution

- Select the minimum cover of sensitive data
- *enclavlow* [1]: an information flow analysis tool
- Sources of sensitive data marked with `sourceMarker`
- Anonymization marked with `sinkMarker`
- Identity functions; expected to be optimized them away by JIT

```
@JECall
static int process(byte[] encrypted) {
  byte sum = 0;
  byte[] password = sourceMarker(new byte[]{1, 2, 3, 4, 5, 6});
  byte[] decrypted = decrypt(password, encrypted);
  for(byte b : decrypted) sum ^= b;
  return sinkMarker(sum);
}
```

[1] coined from the words "enclave" and "flow"

Threat model

- Adversary has no read/write access to enclave code and memory
- Adversary has arbitrary access to any system resource, *including* non-enclave Java and JVM memory
- The actual adversary also has access to hardware resources, including sensor modules, system clock, etc.
- These resources can be used to implement side channel attacks, which are system-dependent, environment-dependent and architecture-dependent
- Example: Timing attack based on CPU branch prediction optimization
- Our adversary model excludes these side channel attacks

Changes on Uranus model

- Applications are expected to run on a Uranus-based JVM.
- Some behaviour disallowed by Uranus assumed permissible under explicit indication
- Assignment to objects outside enclaves
  - Reads are assumed copied into enclave
  - Writes are assumed always leaking
- Assignment to static fields
  - Reads are cloned into enclave
  - Writing enclave-local data may not be expected behaviour
  - Assumed immediate leak
  - Proposed `EnclaveLocal` wrapper type
- Expected to integrate into Uranus for compile-time checking/optimization

Approach

Intuition: Trivial ways of leaking data

```
static int outside; // static variables stored outside enclave
@JOCall void println(int x);

int secret() {
  return sourceMarker(123456);
}

@JECall int foo() {
  store = secret(); // assigning to static field
  println(secret()); // passing secret to a JOCall
  return secret(); // returning secret out of a JECall
}
```

Intuition: Non-trivial ways of leaking data

- Assigning to an outside-enclave object:

```
@JECall void x(Box box) {
  box.value = secret();
}
```

- Leaking control flow into variables:

```
if(secret() >= 3) outside++;
for(int i=0; i < secret(); i++) outside++;
```

- Implicit exceptions:

```
int[3] array = new int[3];
array[secret()] = 1;
```

- Transitive application of above

Flow graph

- Analysis framework: Soot [1]
- Each readable/writable data entry as a node
- $(x, y) \in E \implies$ placement of y outside enclave reduces indistinguishability of x
- $(x, y), (y, z) \in E$: transitive
- $(x, y), (x, z) \in E$: Leaking *either* y *or* z distinguishes x
- No way to represent requirement of *both* y *and* z

Procedure

1. Each method is analyzed independently as a Local Flow Graph (LFG)
2. LFG contracted into subgraph of only "public" nodes called Contract Flow Graph (CFG)
3. CFGs merged together into Aggregate Flow Graph (AFG),

Local flow graph

- Analyze each method independently using Soot's `ForwardBranchedFlowAnalysis`
- Soot calls the `flowThrough` method for each 3AC (jimple) statement
- Each `flowThrough` maps the state in the previous step to a new state
- For branching statements, each branch has a its own state
- When flows converge, soot calls the `merge` method to map two states into a new one

Implementing `flowThrough` for assignment operations

- For each type of value, define
  - rvalue nodes
  - lvalue nodes for assignment
  - and lvalue nodes for deletion
- Pseudocode:

```
Algorithm assign($left = $right):
  for each lvalues($left, FLAG_DELETION) as $node:
    delete (*, $node) from flow graph
  for each lvalues($left) as $leftNode:
    add (CTRL, $leftNode) to flow graph
    for each rvalues($right) as $rightNode:
      add ($rightNode, $leftNode) to flow graph // intuitive
  for each lvalues($right) as $leftNode:
    for each rvalues($left) as $rightNode:
      add ($rightNode.*, $leftNode.*) to flow graph // field projection
```

Global nodes

- Static node
  - All reads use Uranus cache
  - Writes are assumed immediate leak
- Explicit source/sink
  - Explicit sink is cosmetic

Method signature nodes

- Parameter nodes
  - Supports both read and write
- This node
  - just an alternative parameter
- Return node
- Throw node
  - just an alternative return path

The CTRL node

- Tracks what data affect the current flow
- Each branched block has its CTRL node
- $(x, \mathrm{CTRL}) \in E \implies \exists$ predicate p such that the branch represented by CTRL executes if and only if $p(x)$
- $(\mathrm{CTRL}_1, \mathrm{CTRL}_2) \in E$ when CTRL_2 is a subbranch of CTRL_1
- $(\mathrm{CTRL}, y) \in E \implies$ leaking y may distinguish hether CTRL was run
- Merging states: select Lowest Common Ancestor

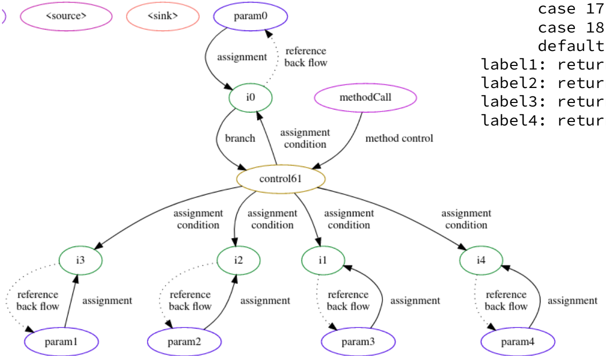
Other nodes

- Local variables
  - Including temp values created by Jimple
- Method calls
  - Blackbox proxies to signature nodes

Case study: Branching (tableswitch)

```java
public static int switchMux(int x,
    int a, int b, int c, int d) {
  switch (x) {
    case 16: return a;
    case 17: return b;
    case 18: return c;
    default: return d;
  }
}
```

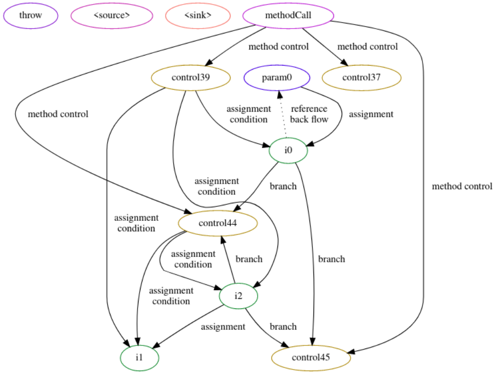
```
public static int switchMux
    (int, int, int, int, int) {
  int i0, i1, i2, i3, i4;
  i0 := @parameter0: int;
  i3 := @parameter1: int;
  i2 := @parameter2: int;
  i1 := @parameter3: int;
  i4 := @parameter4: int;
  tableswitch(i0) {
      case 16: goto label1;
      case 17: goto label2;
      case 18: goto label3;
      default: goto label4; };
label1: return i3;
label2: return i2;
label3: return i1;
label4: return i4; }
```



Case study: Loops

```
public static int loopInc(int i) {
  int a = 0;
  for (int j = 0; j < i; j++) {
      a += j;
  }
  return a;
}
```

```
public static int loopInc(int) {
  int i0, i1, i2;
  i0 := @parameter0: int;
  i1 = 0;
  i2 = 0;
label1:
  if i2 >= i0 goto label2;
  i1 = i1 + i2;
  i2 = i2 + 1;
  goto label1;
label2: return i1; }
```



Contraction and aggregation

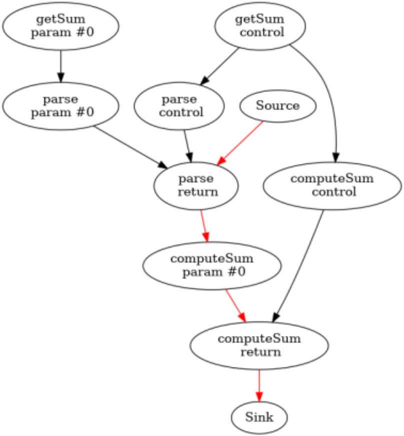
- At each return point, track flow from "public" nodes into the following nodes:
  - `this`
  - `static`
  - explicit sink
  - parameters
  - method calls (proxy nodes)
- At aggregation, connect each proxy node to the actual implementation
  - Polymorphism?

AFG: Full Example

```java
@JECall
int getSum(byte[] enc) {
  byte[] dec = decrypt(enc);
  return sinkMarker(computeSum(dec)); }

byte[] decrypt(byte[] enc) {
  byte[] otp = sourceMarker(PRIVATE_KEY);
  byte[] result = new byte[enc.length];
  for(int i = 0; i < buf.length; i++)
    result[i] = enc[i] ^ otp[i];
  return result; }

int computeSum(byte[] bytes) {
  int result = 0;
  for(byte b : bytes) result += (int) result;
  return result; }
```



Limitations

False negatives: implicit exceptions

- Conditional runtime errors leaks data
  - `NullPointerException`
  - `IndexOutOfBoundsException`
- Too sensitive to raise an exception path for every array access and object access
- Good practice: exceptions should be caught at enclave boundary anyway!
- Solutions usually achieved at the language level, e.g. `@NonNull`, `@Size`
- Do not reinvent the wheel

False positives: identical branches

- This does not leak `secret` (assuming `doSomething*()` do not leak control flow)

```
boolean foo() {
  int secret = getSecret();
  if(secret > 1) {
    doSomething();
    return false;
  } else {
    doSomethingElse();
    return false;
  }
}
```

- Suggested fix: factorize common code out of branches (good code style)

False positives: self-anonymization

- This does not leak `secret` (assuming `doSomethingWith(int)` does not leak)

```
int foo(int a) {
  int secret = getSecret();
  a += secret;
  doSomethingWith(a);
  a -= secret;
  return a;
}
```

- Suggested fix: create a clone of `a`
- Immutability paradigms?
  - We are considering the *compiled binary* layer, so functional programming languages do not help

Tackling polymorphism

- Computing all combinations of instance classes requires exponential time.
- Solution: union of CFGs of all possible subclasses
  - Iago attacks is still possible if code is used elsewhere in enclave
  - Therefore all subclasses, not only those through call analysis, are considered

Tackling JNI calls

- Assumptions:
  - parameters are independent
  - return value is the output
  - control flow is not leaked
- False negative example: `System.arraycopy`
- Uranus turns system calls into `OCall`s, which leak control flow.

Integration into Uranus

- Uranus disallows reads/writes of objects outside enclave without `SafeGetField` etc
- Runtime overhead of checking object location
- *enclavlow* to perform this analysis at compile-time

Conclusion

- Assist with decisions on performance/security tradeoff
- Incorporated into Uranus
- Applications in big data industry
- Room for improvement

References

Einarsson, A., and Nielsen, J. D.
A survivor's guide to java program analysis with soot.
BRICS, Department of Computer Science, University of Aarhus, Denmark 17 (2008).

Jiang, X. J., Tzs, C., Li, O., Shen, T., and Zhao, S.
Uranus: Simple, efficient sgx programming and its applications [unpublished].
In *Proceedings of the 15th ACM ASIA Conference on Computer and Communications Security (ASIACCS '20)* (2020).