SpanDex: Secure Password Tracking for Android

Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Sai Cheemalapati, and Bi Wu

Duke University
Where do your passwords go?
Phishing apps

Fake NetFlix app
From Malware Genome Project

Sends passwords to
http://erofolio.no-ip.biz/login.php
Phishing apps

Wroba (Korean malware) Svpeng (Russian malware)

“Let’s use taint tracking!”
General approach

1. Tag password as entered
General approach

1. Tag password as entered
General approach

1. Tag password as entered
General approach

1. Tag password as entered
General approach

1. Tag password as entered

ScreenPass [MobiSys ‘13] Spoof-resistant UI for entering passwords
General approach

1. Tag password as entered
2. Track tags as app runs
3. Inspect output tags

TaintDroid [OSDI ‘10] tracks how data flows through Android apps
Taint-tracking basics

c ← a op b  
\[
\text{taint}(c) \leftarrow \text{taint}(a) \cup \text{taint}(b)
\]

Each variable has a label/tag; Labels reflect data dependencies
Taint-tracking basics

\[ c \leftarrow a \text{ op } b \quad taint(c) \leftarrow taint(a) \cup taint(b) \]
Taint-tracking basics

c ← a op b

taint(c) ← taint(a) ∪ taint(b)

setTaint(a, t)

taint(a) ← \{t\}

c = a + b

taint(c) ← \{t\} ∪ \{\} = \{t\}

Explicit flow
Directly transfers information from source to destination
Taint-tracking basics

$c \leftarrow a \text{ op } b \quad \text{taint}(c) \leftarrow \text{taint}(a) \cup \text{taint}(b)$

$\text{setTaint}(a, t) \quad \text{taint}(a) \leftarrow \{t\}$
$c = a + b \quad \text{taint}(c) \leftarrow \{t\} \cup \{\} = \{t\}$

$\text{if } (c == 0)$
$s = 1$

Implicit flow
Information transferred via control flow
Tracking implicit flows is (really) hard.
Taint the PC

if (s == 0) {
    x = a
} else {
    y = b
}

output x
output y

taint(\text{PC}_{\text{curr}}) \leftarrow taint(s) \cup taint(\text{PC}_{\text{prev}})

taint(x) \leftarrow taint(a) \cup taint(\text{PC}_{\text{curr}})

taint(\text{PC}_{\text{curr}}) \leftarrow taint(\text{PC}_{\text{prev}})

Tainting the PC captures Information flow into x
Taint the PC

```python
if (s == 0) {
    x = a
} else {
    y = b
}
output x
output y
```

Problem: y contains same secret information as x, even though it wasn’t updated
Bigger problem: overtainting

```java
if (s == 0) {
    // complex block of code
}
```

Condition may reveal very little secret information

Taint tags updated as if objects contain all secret information
Not much information transferred to a large number of objects.
Problem: overtainting

TaintDroid
(explicit flows)

TaintDroid++
(explicit flows, tainted PC)
Problem: overtainting

TaintDroid
(explicit flows)

TaintDroid++
(explicit flows, tainted PC)
Key observation

```java
if (s == 0) {
    x = a
    ...
} else {
    y = b
    ...
}
```

If OK to leak that s!=0, then don’t propagate taint

At most, reveals whether s is 0
Our solution: SpanDex

- **Tracks implicit flows within Dalvik VM**
  - Can compute a *useful* upper bound on info leaks
- **Leverages key properties of passwords**
  - Short strings
  - Never displayed on screen
  - Limited local processing
SpanDex overview

1. Initialize **possibility set (p-set)** for taint source
   - [32, 126] for each password character

2. **Record operations performed on tainted data**
   - Operations recorded in **Operation DAG (op-DAG)**

3. **Update p-set when involved in branch condition**
   - op-DAG + branch conditions → CSP

4. **Guarantee for untainted outputs**
   - Leak at most as much info as reflected in p-sets
   - Allows for rich set of policies for limiting leaks
High-level example

```
// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' &&
    c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"
```
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' &&
    c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' && c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"

p-set: [32, 126]
size: 95

INIT_PSET(c, [32,126])
High-level example

// password input ‘P’
initPset(c, PASSWORD)
// end password input

if (c >= 'A' && c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output “value was P”

INIT_PSET(c, [32, 126])
LOG_CMP(c>=65, T)

p-set: [65, 126]
size: 62
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' &&
    c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"

INIT_PSET(c, [32, 126])
LOG_CMP(c>=65, T)
LOG_CMP(c<=90, T)

p-set: [65, 90]
size: 26 (uppercase letters)
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' && c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"

p-set: [65, 90]
size: 26 (uppercase letters)
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' && c <= 'Z')
    lc = c + 32

if (lc == 'p')
    output "value was P"

p-set: [80]
size: 1 ('P')
High-level example

// password input 'P'
initPset(c, PASSWORD)
// end password input

if (c >= 'A' && c <= 'Z')
  lc = c + 32

if (lc == 'p')
  output "value was P"

p-set: [80]
size: 1 (‘P’)
Lower-level example

Explicit flows create new op-DAG nodes

Conditional branches require solving CSP to update p-set(s)

```
0000: mov v1, v0  // v0, v1 label=N0
0002: add v2, v1, 3   // v2's label=N1
0004: add v2, v2, 2   // v2's label=N2
0006: sub v3, 6, v2   // v3's label=N3
0008: add v2, v2, 7   // v2's label=N4
000a: const/16 v4, 122 // v4's label=0
000c: if-le v3, v4, 0016
000e: ...
```
Lower-level example

0000: mov v1, v0  // v0, v1 label=N0
0002: add v2, v1, 3  // v2's label=N1
0004: add v2, v2, 2  // v2's label=N2
0006: sub v3, 6, v2  // v3's label=N3
0008: add v2, v2, 7  // v2's label=N4
000a: const/16 v4, 122  // v4's label=0
000c: if-le v3, v4, 0016
000e: ...

CSP solver traverses op-DAG
back to root ➔
v0 + 6 - 2 - 3 ≤ 122
Other considerations

• **CSPs may hard to solve**
  – CSP may involve multiple sources (e.g., pw chars)
  – CSP may involve complex operations (e.g., bitwise)
  – We see this in crypto and string-encoding libraries

• **Solution**
  – Define a set of trusted runtime libraries
  – No CSP-solving internally
  – Taint all trusted-lib outputs
  – Ban complex operations in untrusted code

• **More details in paper**

Constrains how apps operate on secret data (e.g., must use trusted crypto lib)
SpanDex evaluation

- What is SpanDex’s runtime overhead?
- What p-sets do we observe in real apps?
- How well does SpanDex protect passwords?
SpanDex evaluation

• What is SpanDex’s runtime overhead?
• How do apps update p-sets?
• How well does SpanDex protect passwords?
Attacker model

Assume target’s password is in database

Large DB of plaintext passwords

Attacker can leak username + p-set of each pw character + pw length
Attack simulation

• Assume attacker learns each character’s type
  – Lower case (a-z) or
  – Upper case (A-Z) or
  – Numeric (0-9) or
  – Special (!@#$ …)

• How many guesses would attacker need?
  – Assume online querying
  – Hope that number of guesses is large
Attack simulation

• Dataset
  – DB of 131 million unique passwords
  – Collected from a variety of well known leaks

• Procedure
  – For each password, P, in DB
  – Generate rule describing each char’s type
  – **Match set** := set of passwords that match P’s length, char types
  – Match set is set of all possible passwords that could be P
  – Want to know, for each P, how large is its match set?
Match-set size distribution

96% of passwords have $|\text{match set}| > 1k$

98% of passwords have $|\text{match set}| > 100$

Interesting implication of attacker model: longer passwords are less secure
Match-set size distribution

But what if some passwords are more likely than others? What is the expected number of tries before a successful guess?
Uniform password usage

>10 tries for 99% of passwords

>100 tries for 90% of passwords
Bad Zipf-like password usage

90% of passwords in <50 tries

80% of passwords in <50 tries

Percentage

Expected number of tries

s=1
s=0.7878 (rockyou)
s=0.695 (flirtlife)
Better Zipf-like password usage

CDF of expected tries

Percentage

Expected number of tries (logsacle)

>10 tries for 99% of passwords

>100 tries for 85% of passwords

s=0.246 (hotmail)
s=0.23 (c-bits)
Better Zipf-like password usage

Expected number of tries largely depends on usage distribution. A wide range have been observed practice.

s=0.246 (hotmail)
s=0.23 (c-bits)
Related work

• Dynamic tracking for implicit flows
  – Dytan [Clause ‘07], DTA++ [Kang ‘11]

• Quantifying revealed info
  – FlowCheck [McCamant ‘08]

• Process-level tracking
  – Asbestos [Efstathopoulos ‘05],
    HiStar [Zeldovich ‘06], Flume [Krohn ‘07]

• Symbolic execution
SpanDex

• p-sets give upper bound on implicit leaks
  – Can track in real-time
  – Rich policy possibilities

• Useful under specific conditions
  – We haven’t “solved” the implicit-flow problem
  – Requires simple processing of secret data

• Future
  – Can look at other types (e.g., CCNs, SSNs)
  – Runtime CSPs limitations may be useful
Runtime performance

• Runtime overhead, no sensitive data:
  – 16% vs 10% for TaintDroid

• Time to handle branch on sensitive data:
  – < 0.1ms for logs up to 100 arith. Ops
  – Log length in practice: avg: 2 ops, max: 93 ops
  – Rate of tainted branches: ~100s/min
  – Expect to spend a few ms per sec updating p-sets

Summary: can track p-sets in real-time