With a Case Study on Binary Vulnerability Analysis

Cyber-security: the Journey from Formal Methods, Program Analysis to Data Analytics

LIU, Yang

ICECCS 2017 (8 Nov 2017)
Since 2005

Formal Methods
Hello, This is Tom

Hello, Prove it!

This is the proof

\[
\begin{align*}
\alpha_1 & : A \rightarrow I : A.I.\{N_a \cdot A\}_{K_I} \\
\beta_1 & : I(A) \rightarrow B : A.B.\{N_a \cdot A\}_{K_B} \\
\beta_2 & : B \rightarrow I(A) : B.A.\{N_a \cdot N_b\}_{K_A} \\
\alpha_2 & : I \rightarrow A : I.A.\{N_a \cdot N_b\}_{K_A} \\
\alpha_3 & : A \rightarrow I : A.I.\{N_b\}_{K_I} \\
\beta_3 & : I(A) \rightarrow B : A.B.\{N_b\}_{K_B}
\end{align*}
\]
A self-contained framework to support the development of formal verification tools

- Design verification
  - (Timed) Security Protocol (FM15,TSE17)
  - TPM Verification (FM14)

- Implementation verification
  - Authentication Protocols (NDSS13)
  - Android App for Malware (TSE17)

- Assembly code verification
  - Vulnerability Verification
Since 2015

Securify: A Compositional Approach of Building Security Verified System
**Verification**

- To overcome security issues by deploying composition based techniques to realize security-verified systems
- To introduce runtime verification to further improve robustness
Proposed Approach

Securify Architecture

- Application
- Verified Libraries
- Guest OS
- HAL
- Drivers
- System Monitor Partition
- System Driver Partition
- System Partition

Verified Separation Micro-kernel
- XtratuM
  - Health Monitoring
  - Partition Management
  - Drivers/Trap Management
  - Kernel Security Functions

Verified Leon4 SoC
- MMU
- I/O MMU

SP8: Compositional Security Reasoning with Untrusted Components (ASE 15)

SP7: Model-based Secure Code Generation

SP6: Automatic Program Verification (CAV 17)

SP5: Security-Enhanced Library Verification (CAV 16)

SP3: Secure Micro-Kernel Verification (TACAS16,17, TDSC17)

SP4: Runtime Security Verification (FM 15)

SP2: Hardware-aided Dynamic Security Analysis (DAC17, TIFS16)

SP1: Hardware Verification (FM 16)
SP1: Hardware Verification

SP2: Hardware-aided Dynamic Security Analysis

SP3: Secure Micro-Kernel Verification

SP4: Runtime Security Verification

SP5: Security-Enhanced Library Verification

SP6: Automatic Program Verification

SP7: Model-based Secure Code Generation

SP8: Compositional Security Reasoning with Untrusted Components

Team Members:

- Srikanthan Thambipillai (NTU)
- Liu Yang (NTU)
- Alwen Tiu (NTU)
- Chin Wei Ngan (NUS)
- Sun Jun (SUTD)
- Dong Jin Song (NUS)
- Ralf Huuck (NICTA)
- David Basin (ETH)
- Kenny Paterson (RHUL)
- Sjouke Mauw (Uni. of Lux.)
- Luke Ong (Oxford)
- Wei Zhang (HKUST)

Sponsors:

- ST Electronics
- Wincor Nixdorf
Is FM really useful for solving real-world security problems?
Case Study: Binary Vulnerability Detection?

- No need to rely on compilation process
- Vendors need not release source code
- Cross-architecture and cross platform analysis

Methods for detecting vulnerabilities at the binary level!
LLVV: Automatic Vulnerability Verification for LLVM’s Intermediate Representation, 2014
Detection of vulnerabilities sometimes relies upon partial semantics of variables rather than their values.

Examples:
- `strcpy(des, src) → Len(src) + 1 > size(des)`
- `memcpy(des, src, num) → value(num) > size(des)`

LLVV: Automatic Vulnerability Verification on LLVM IR
- Context- and path-sensitive
- Language independent
Approach Overview

Binary Executables → LLVM IR Disassembler → LLVM Intermediate Representation →

ICFG Construction → Pointer Analysis → API/Library Translation → Instruction Translation →

Library Call Summarizations → Vulnerabilities VDSL Patterns →

Weighted Pushdown System Model Generator → WPDS Model

- Detected Vulnerabilities (with witness)
- Proved

WPDS Model Checker →

Test Case Generator

Test Case → LLVM Execution Engine →

Vulnerabilities Simulation Report
Let \( \text{Attr} = \{\text{Size}, \text{Len}, \text{Val}, \text{Type}\} \) denote size, length, value and type attributes. Other demanded attributes can be easily introduced. VDSL formulas are defined by the following syntax:

\[
\phi ::= f(S1, ..., Sk) \land \psi \\
\psi ::= \text{exp} \Rightarrow \text{exp} \mid \psi \land \psi \mid \psi \lor \psi \\
\Rightarrow ::= \leq \mid \geq \mid < \mid > \mid == \mid \neq \\
\text{exp} ::= \text{attr}(i) \mid i \mid \text{exp} \text{op} \text{exp} \\
\text{op} ::= + \mid - \mid \times \mid / \mid \%
\]

where \( f \) is a function, \( i, k \) are integers, \( \text{attr} \in \text{Attr} \).

E.g., \( S1 \Rightarrow a, S2 \Rightarrow \text{argv}[1] \)
\[
\text{strcpy}(S1, S2) \land \text{Size}(S1) < \text{Len}(S2)
\]

**Specifying constraints on vulnerabilities over attributes**

**Null Pointer Dereference Vulnerability Pattern**

\[\text{Assign}_1(S1, \ast S2) \land \text{Val}(S2) = 0.\]

**Division by Zero Vulnerability Pattern**

\[\text{Assign}_3(S1, S2, S3) \land \text{Val}(S3) = 0.\]
Not working
Three main problems in applying FM

- Models
- Scalability
- Domain Knowledge
Cross-Architecture Cross-OS Binary Search

Why?

- Plagiarism Detection, Clone detection, Vulnerability extrapolation, Search engine for machine code, Code property inference, Type inference, Partial decompilation, Malware signature generation, Vulnerability signature generation

Reality: The Heartbleed Example

- Same program (at source code level) might look totally different at machine code level

```c
leax edx, [ebx+18h]
add edx, 8
mov [esp+4Ch+var_20], edx
mov [esp+4Ch+dest], edx
mov [esp+4Ch+n], 0A14h
mov [esp+4Ch+src], offset aT1_lib_c; "t1_lib.c"
call CRYPTO_malloc
mov byte ptr [eax], 2
mov ebp, eax
mov eax, ebx
shr eax, 8
mov [ebp+2], bl
leax edx, [ebp+3]
mov [ebp+1], al
mov [esp+4Ch+n], ebx; n
mov [esp+4Ch+dest], edx; dest
mov [esp+4Ch+var_24], edx
mov [esp+4Ch+src], edi; src
call _memcpy
mov edx, [esp+4Ch+var_24]
mov [esp+4Ch+src], 10h
add ebx, edx
mov [esp+4Ch+dest], ebx
call RAND_pseudo_bytes
mov edx, [esp+4Ch+var_20]
mov [esp+4Ch+n], ebp
mov [esp+4Ch+src], 10h
mov [esp+4Ch+dest], esi
mov [esp+4Ch+var_40], edx
call ss13_write_bytes
test eax, eax
jz short loc_6087920
```

GCC

```c
leax edx, [ebx+18h]
add edx, 8
mov [esp+4Ch+var_20], edx
mov [esp+4Ch+dest], edx
mov [esp+4Ch+n], 0A14h
mov [esp+4Ch+src], offset aT1_lib_c; "t1_lib.c"
call CRYPTO_malloc
mov byte ptr [eax], 2
mov ebp, eax
mov eax, ebx
shr eax, 8
mov [ebp+2], bl
leax edx, [ebp+3]
mov [ebp+1], al
mov [esp+4Ch+n], ebx; n
mov [esp+4Ch+dest], edx; dest
mov [esp+4Ch+var_24], edx
mov [esp+4Ch+src], edi; src
call _memcpy
mov edx, [esp+4Ch+var_24]
mov [esp+4Ch+src], 10h
add ebx, edx
mov [esp+4Ch+dest], ebx
call RAND_pseudo_bytes
mov edx, [esp+4Ch+var_20]
mov [esp+4Ch+n], ebp
mov [esp+4Ch+src], 10h
mov [esp+4Ch+dest], esi
mov [esp+4Ch+var_40], edx
call ss13_write_bytes
test eax, eax
jz short loc_6087920
```

Mingw32
P1. Resilient to the syntax and structural gaps introduced due to architecture, OS and compiler differences.

P2. Accurate by considering the complete function semantics.

P3. Scalable to large size real world binaries.

Table 1: Comparison of existing techniques. Here, ✓ denotes limited support, while ✓ and ✗ represent full and no support, respectively.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Technique</th>
<th>P1 Resilient</th>
<th>P2 Accurate</th>
<th>P3 Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracy [10]</td>
<td>Static</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>CoP [26]</td>
<td>Static</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Bug search [29]</td>
<td>Static</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DISCOVRE [35]</td>
<td>Static</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>BLEX [14]</td>
<td>Dynamic</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>BinGo</td>
<td>Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Cross-architecture Cross Platform Vulnerability Mining

Smart Mining Learning + Deep Program Analysis

Binary → Pre-processing → Selective Inlining → Filtering

Trace Pruning

Ranks → Similarity Matching → Function Model Generation → Partial Trace Generation

intel  ARM  Windows  Linux
Two zero-day vulnerabilities (RCE),

Nine zero-day vulnerabilities (DoS, info leakage)

zero-day vul.: unknown vul.
DoS: Deny of Service
RCE: remote code execution
Binary Vulnerability Analysis

Patch Analysis → Vulnerability Database → Binary Function Matching → Vulnerability Report

Target Binary: X-Arch/X-OS
Towards Understanding the Pain and Pills

SPAIN: Security Patch Analysis for Binaries

Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu and Fu Song
ICSE 2017
**Motivating Example**

Goal: to capture the patch location and summarize patch/vulnerability patterns at binary level

---

**Original version**

- `BB_1`
  - `call BN_new`
  - `1: mov edi, eax`
    - `...`
  - `2: cmp eax, [edi+8]`
    - `...`

**Patched version**

- `BB_1`
  - `call BN_new`
  - `1: test eax, eax`
    - `mov edi, eax`
    - `jz error`
  - `BB_3`
    - `error: ...`

- `BB_2`
  - `...`
  - `c: cmp eax, [edi+8]`
    - `...`

---

a: Additional instruction to test the value of a register “eax”

b: Additional flow branch to error
A scalable binary-level patch analysis framework, SPAIN, to automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns.
Step 3: Identifying Security Patches

- Assumption: a security patch is less unlikely to change the semantics

- Trace sem summary = post-state – pre-state

```
mov eax, 0x04
sub ebx, eax
```
(a) Sample Code Segment

```
Pre-state:
Reg = \{eax = 0, ebx = 0, ..\}  \quad \text{Reg}' = \{eax' = 0x4, ebx' = -0x4, ..\}
Flag = \{zf = 0, sf = 0, ..\}  \quad \text{Flag}' = \{zf' = 0, sf' = 1, ..\}
Mem = \{0, 0\ldots 0\}  \quad \text{Mem}' = \{0, 0\ldots 0\}
```
(b) Pre- and Post-State before and after Executing the Code Segment

- Identify the overall semantic change

  Sem Diff1: a->b’->c vs a->b->c
  Sem Diff2: a->b’->d vs a->b->d
  Sem Diff3: a->b’->g’->d vs a->b->d

(a) Original Function  (b) Patched Function
We manually identified all the security and non-security patches of all the 20 versions of OpenSSL 1.0.1 (446,747 LOC) by analyzing its commits on GitHub (63 CVEs)
Linux Kernel is an open source software with around 18,963,973 LOC and developed since 2002.

Adobe PDF Reader is a closed source software. We use two of its libraries, 3difr.x3d and AXSLE.dll, which have around 1,293 and 4,874 functions respectively.

<table>
<thead>
<tr>
<th>Version</th>
<th>Total Func.</th>
<th>Sec. Patched</th>
<th>T. (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux: 3.16.2 – 3.16.3</td>
<td>249341</td>
<td>1221</td>
<td>807</td>
</tr>
<tr>
<td>difr.x3d: 11.0.08 – 11.0.09</td>
<td>1293</td>
<td>12</td>
<td>21</td>
</tr>
<tr>
<td>difr.x3d: 11.0.13 – 11.0.14</td>
<td>1293</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>difr.x3d: 11.0.15 – 11.0.16</td>
<td>1293</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>AXSLE.dll: 11.0.15 – 11.0.17</td>
<td>4875</td>
<td>27</td>
<td>84</td>
</tr>
</tbody>
</table>
## Evaluation: Patch & Vulnerability Patterns

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Concrete Vulnerability</th>
<th>Vulnerability Pattern</th>
<th>Concrete Patch</th>
<th>Patch Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Free</td>
<td>call BN_to.ASCIIINTEGER test eax, eax mov esi, eax mov [esp + 4Ch + var_4C], esi call ASCII_STRING_clear_free mov [esp + 4Ch + var_4C], esi call ASCII_STRING_clear_free</td>
<td>call (untrusted_func) (sanity check) : (return_value) ... mov (func_param), (return_value) call (free) ... mov (func_param), (return_value) call (free)</td>
<td>call BN_to.ASCIIINTEGER test eax, eax mov esi, eax mov [esp + 4Ch + var_4C], esi call ASCII_STRING_clear_free</td>
<td>call (untrusted_func) (sanity check) : (return_value) ... mov (func_param), (return_value) call (free) ...</td>
</tr>
<tr>
<td>Integer Underflow</td>
<td>mov eax, [esp + 4Ch + arg_4] ... mov ecx, eax sub ecx, [esp + 4Ch + var_2C] add [esp + 4Ch + var_24], ecx mov esi, [esp + 4Ch + _24] mov ecx, [esp + 4Ch + arg_8] mov [ecx], esi</td>
<td>mov (TNT_IN), (untrusted_src) (arith_op) : (TNT_result), (TNT_IN) mov (sec_SSTV_sink), (TNT_result)</td>
<td>mov eax, [esp + 4Ch + arg_4] ... mov edi, [esp + 4Ch + var_2C] cmp eax, edi jle error mov ecx, eax sub ecx, [esp + 4Ch + var_2C] add [esp + 4Ch + var_24], ecx mov esi, [esp + 4Ch + _24] mov ecx, [esp + 4Ch + arg_8] mov [ecx], esi</td>
<td>mov (TNT_IN), (untrusted_src) (sanity check) : (TNT_IN) ... (arith_op) : (TNT_result), (TNT_IN) mov (sec_SSTV_sink), (TNT_result)</td>
</tr>
<tr>
<td>Use After Free</td>
<td>mov ebp, [esp + 0ECH + arg_0] ... mov [esp + 0ECH + dest], ebp call ssl3_release_read</td>
<td>mov (TNT_IN), (untrusted_src) mov (func_param), (TNT_pointer) call (free)</td>
<td>mov ebp, [esp + 0ECH + arg_0] ... mov eax, [ebp + 58h] test eax, eax jnz (do not release) mov [esp + 0ECH + dest], ebp call ssl3_release_read</td>
<td>mov (TNT_IN), (untrusted_src) (sanity check) : (TNT_pointer) mov (func_param), (TNT_pointer) call (free)</td>
</tr>
<tr>
<td>NULL Pointer Dereference</td>
<td>call BN_new mov edi, eax cmp eax, [edi + 8]</td>
<td>call (untrusted_func) (mem_deref) : (return_value)</td>
<td>call BN_new mov edi, eax mov edi, eax jnz error ... cmp eax, [edi + 8]</td>
<td>call (untrusted_func) (mem_deref) : (return_value)</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>mov ebx, [esp + 3Ch + arg_8] mov [esp + 3Ch + n], ebx call eax</td>
<td>mov (TNT_IN), (untrusted_src) mov (func_param), (TNT_IN) call (untrusted_func)</td>
<td>mov ebx, [esp + 3Ch + arg_8] jmp ebx, [esp + 3Ch + arg_4] jle error ... mov [esp + 3Ch + n], ebx call eax</td>
<td>mov (TNT_IN), (untrusted_src) (sanity check) : (TNT_IN) mov (func_param), (TNT_IN) call (untrusted_func)</td>
</tr>
</tbody>
</table>

Details can be found at: [https://sites.google.com/site/binaryanalysisicse2017/claim/patterns](https://sites.google.com/site/binaryanalysisicse2017/claim/patterns)
Binary Vulnerability Analysis

- Patch Analysis
- Target Binary
- Binary Function Matching
- Vulnerability Database
- Input (seeds)
- Fuzzing
- X-Arch/X-OS
- Linux/Windows
- Vulnerability Report
Skyfire: Data-Driven Seed Generation for Fuzzing

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu
S&P 2017
Fuzzing Target: Stages of processing structured inputs

- Syntax parsing
- Semantic checking
- Execution

Structured inputs

Parsing errors → Semantic violations → Crashes → Successful executed
## Passing syntax parsing/semantic checking

<table>
<thead>
<tr>
<th></th>
<th>Grammar</th>
<th>manually-specified generation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax rules</td>
<td>easy</td>
<td></td>
</tr>
<tr>
<td>semantic rules</td>
<td>hard</td>
<td></td>
</tr>
</tbody>
</table>

**hard:**
- different programs may implement different sets of semantic rules,
- daunting, labor-intensive, or even impossible

- Design goal: generates well-distributed test cases
  - Capture the syntax rules and valid constants
  - Extend context-free grammar (CFG) to model semantic rules
**Probabilistic context-sensitive grammar**

**Context-free grammar (CFG)** $G_{cf} = (N, \Sigma, R, s)$:
- $N$ is a finite set of non-terminal symbols,
- $\Sigma$ is a finite set of terminal symbols,
- $s \in N$ is a distinguished start symbol.
- $R$ is a finite set of production rules of the form $\alpha \rightarrow \beta_1\beta_2...\beta_n$, $\alpha \in N$, $n \geq 1$, $\beta_i \in (N \cup \Sigma)$ for $i = 1...n$,

**Context-sensitive grammar (CSG)** $G_{CS} = (N, \Sigma, R, s)$:
- $[c]\alpha \rightarrow \beta_1\beta_2...\beta_n$,

<typename of $\alpha$'s great-grandparent, type of $\alpha$'s grandparent, type of $\alpha$'s parent, value of $\alpha$'s first sibling or type of $\alpha$'s first sibling if the value is null>

**Probabilistic context-sensitive grammar (PCSG)** $G_p = (G_{CS}, q)$,

$q : R \rightarrow R^+, \forall \alpha \in N : \Sigma[c] \alpha \rightarrow \beta_1\beta_2...\beta_n \in R \cdot q([c] \alpha \rightarrow \beta_1\beta_2...\beta_n) = 1$. 
nodes 5, 14:
attribute $\rightarrow$ version="1.0", c=[null, document, prolog, <?xml]
# Learned production rules of XSL

<table>
<thead>
<tr>
<th>Context</th>
<th>Production rule</th>
<th>Prob.</th>
</tr>
</thead>
</table>
| [null,null,null,null]         | document $\rightarrow$ prolog element  
                               |         | 0.8200  |
|                               | element                                                                         |         | 0.1800  |
| [null,null,document,null]     | prolog $\rightarrow$ <?xml attribute attribute?> 
                               |         | 0.6460  |
|                               | $\rightarrow$ <?xml attribute?>                                                 |         | 0.3470  |
|                               | $\rightarrow$ ...                                                               |         |         |
| [null,null,document,prolog]   | element $\rightarrow$ <xsl:stylesheet attribute attribute attribute>content<xsl:stylesheet>  
                               |         | 0.0034  |
|                               | $\rightarrow$ <xsl:transform attribute attribute>content<xsl:transform>         |         | 0.0001  |
|                               | $\rightarrow$ ...                                                               |         |         |
| [document,element,content,element] | element $\rightarrow$ <xsl:template attribute>content<xsl:template>  
                               |         | 0.0282  |
|                               | $\rightarrow$ <xsl:variable attribute>content<xsl:variable>                    |         | 0.0035  |
|                               | $\rightarrow$ <xsl:include attribute/>                                         |         | 0.0026  |
|                               | $\rightarrow$ ...                                                               |         |         |
| [null,document,prolog,<?xml>] | attribute $\rightarrow$ version="1.0"                                         |         | 0.0056  |
|                               | $\rightarrow$ encoding="utf-8"                                                 |         | 0.0021  |
|                               | $\rightarrow$ ...                                                               |         |         |
Left-most derivation

t0=document
t1=prolog element
t2=<xml attribute attribute?> element
t3=<xml version="1.0" attribute?> element
t4=<xml version="1.0" encoding="utf-8">element
t5=<xml version="1.0" encoding="utf-8"> <xsl:stylesheet attribute>content</xsl:stylesheet>
t6=<xml version="1.0" encoding="utf-8"> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">content</xsl:stylesheet>
t7=<xml version="1.0" encoding="utf-8"> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">element</xsl:stylesheet>
t8=<xml version="1.0" encoding="utf-8"> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:output attribute/></xsl:stylesheet>
t9=<xml version="1.0" encoding="utf-8"> <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"> <xsl:output xsl:use-attribute-sets=""/>
</xsl:stylesheet>

<xml version="1.0" encoding="utf-8"> 
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output xsl:use-attribute-sets=""/>
</xsl:stylesheet>
Experiment setup

Sablotron
  - Adobe PDF Reader and Acrobat.

libxslt
  - Chrome browser, Safari browser, and PHP 5

libxml2
  - Linux, Apple iOS/OS X, and tvOS
## Bugs found in XSLT and XML engines

<table>
<thead>
<tr>
<th>Unique Bugs (#)</th>
<th>XSL</th>
<th>XML</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sablotron 1.0.3</td>
<td>libxslt 1.1.29</td>
</tr>
<tr>
<td>Memory Corruptions (New)</td>
<td>Crawl+AFL</td>
<td>Skyfire</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Memory Corruptions (Known)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Denial of Service (New)</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Total</td>
<td>9</td>
<td>13</td>
</tr>
</tbody>
</table>

† CVE-2015-7115, CVE-2015-7116, CVE-2016-1835, CVE-2016-1836, CVE-2016-1837, CVE-2016-1762, and CVE-2016-4447; pending reports include GNOME bugzilla 766956, 769185, 769186, and 769187.
‡ CVE-2012-1530, CVE-2012-1525.
⊕‡ GNOME bugzilla 759579, 759495, and 759675.

Skyfire: inputs generated by Skyfire
Crawl+AFL: Fed the samples crawled as seeds to AFL
Skyfire+AFL: the inputs generated by Skyfire as seeds to AFL
## Vulnerabilities and Types

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2016-6978</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>CVE-2016-6969</td>
<td>Use-after-free</td>
</tr>
<tr>
<td>Pending advisory 1</td>
<td>Double-free / UAF</td>
</tr>
<tr>
<td>CVE-2017-2949</td>
<td>Out-of-bound write</td>
</tr>
<tr>
<td>CVE-2017-2970</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>CVE-2015-7115</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>CVE-2015-7116</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>CVE-2016-1832</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>CVE-2016-1836</td>
<td>Use-after-free</td>
</tr>
<tr>
<td>CVE-2016-1837</td>
<td>Use-after-free</td>
</tr>
<tr>
<td>CVE-2016-4447</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>Pending advisory 2</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>Pending advisory 3</td>
<td>Out-of-bound read</td>
</tr>
<tr>
<td>Pending advisory 4</td>
<td>Use-after-free</td>
</tr>
<tr>
<td>Pending advisory 5</td>
<td>Out-of-bound read</td>
</tr>
</tbody>
</table>

We discovered 19 new memory corruption bugs (among which we discovered 16 new vulnerabilities and received 33.5k USD bug bounty rewards) and 32 denial-of-service bugs.
On-going Testing Works

- Symbolic Execution (Whitebox)
  - Loop (FSE 16, FSE 17, ASE 17), API summary and SMT

- Good Test Cases (Blackbox)
  - (Systematic) Test Case Generation
  - Model Based Testing:
    - FSE 2017: Guided, Stochastic Model-Based GUI Testing of Android Apps

- Feedback Based Testing (Greybox)
  - Improve the feedback
    - FSE 2017: Steelix: Program-State Based Binary Fuzzing
  - Runtime Seed Prioritization

- Combining Different Ideas
  - Testing Orchestration
    - Static Analysis, Random Testing, Taint + Machine Learning

Mobile Performance Testing
Security Protocol Fuzzing
Android OS Fuzzing and Attack Generation
Robot and Automotive Testing
Binary Vulnerability Analysis

- Patch Analysis
- Target Binary
- Binary Function Matching
- Vulnerability Database
- Fuzzing
- Crash Root Cause Analysis
- Vulnerability Model
- Language Grammar
- Input (seeds)
- Static Analysis
- Dynamic Analysis
- Machine Learning
- Data Analytics

X-Arch/X-OS

Static Analysis
Dynamic Analysis
Machine Learning
Data Analytics
Summary of the Ideas

Correctness
Security
Reliability
Performance
Robustness

Analyzing Complex Systems

Formal Models and Precise Semantics

Formal Analysis
Program Analysis
Machine Learning
Data Analytics
Similar Ideas on Malware Analysis

Malware Model

Generation

Detection & Classification

Learning
Similar Ideas on Security Protocol Analysis
Security Research

- Runtime Adaptive Security (Trust + Resilient)
  - Runtime attack monitoring (Logic and physical model based)
  - Dynamic Adaptation (using based on attack-defence tree and game theory) for ROS
  - Security verification and resilient guarantee
  - Platforms: IT architecture, AV/Drones, IoT/urban computing/smart nation
  - Next generation of security operation center

- Mobile security: malware and vulnerability detection, trend, attribution...

- Binary Analysis
  - Vulnerability learning, matching and fuzzing
  - Patch Analysis and Summary
  - Crash root-cause analysis and debugging
  - Binary repairing and hardening
  - Binary reverse engineering

- Security Verification

- Security using HW and Hardware Security

- Network Security
Thank you

Whatever you learned will be useful somehow somewhere