Inside AVM

Haifei Li, security researcher
haifeil@microsoft.com

REcon 2012, Montreal

Background

- Last year at CSW, I presented research on exploiting a vulnerability class inside AVM called "*JIT type confusion*". The motivation for that research was that the exploitation hadn't been well-studied.

- The motivation for this research is that the root cause of such vulnerabilities isn't well-studied.

- Since my CSW presentation, we've continued to see Flash as a cause of concern for online threat research (vulnerabilities, zero-day attacks, etc.).

FLASH PLAYER

Version 11.x

Brief	Originally Posted	Last Updated
APSB12-09 Security update available for Adobe Flash Player	5/4/2012	5/4/2012
APSB12-07 Security update available for Adobe Flash Player	3/28/2012	4/5/2012
APSB12-05 Security update available for Adobe Flash Player	3/5/2012	3/5/2012
APSB12-03 Security update available for Adobe Flash Player	2/15/2012	2/15/2012
APSB11-28 Security update available for Adobe Flash Player	11/10/2011	11/10/2011

Version 10.x

Brief	Originally Posted	Last Updated
APSB12-09 Security update available for Adobe Flash Player	5/4/2012	5/4/2012
APSB12-07 Security update available for Adobe Flash Player	3/28/2012	4/5/2012
APSB12-05 Security update available for Adobe Flash Player	3/5/2012	3/5/2012
APSB12-03 Security update available for Adobe Flash Player	2/15/2012	2/15/2012
APSB11-28 Security update available for Adobe Flash Player	11/10/2011	11/10/2011
APSB11-26 Security updates available for Adobe Flash Player	9/21/2011	9/21/2011
APSB11-21 Security updates available for Adobe Flash Player	8/9/2011	8/9/2011
APSB11-18 Security update available for Adobe Flash Player	6/14/2011	6/15/2011
APSB11-13 Security update available for Adobe Flash Player	6/5/2011	6/14/2011
APSB11-12 Security update available for Adobe Flash Player	5/12/2011	6/14/2011
APSB11-07 Security update available for Adobe Flash Player	4/15/2011	4/28/2011
APSA11-02 Security Advisory for Adobe Flash Player, Adobe Reader and Acrobat	4/11/2011	4/28/2011
APSB11-05 Security update available for Adobe Flash Player	3/21/2011	3/21/2011
APSA11-01 Security advisory for Adobe Flash Player, Adobe Reader and Acrobat	3/14/2011	3/21/2011

Bulletins released by Adobe in the past year to address Flash vulnerabilities.

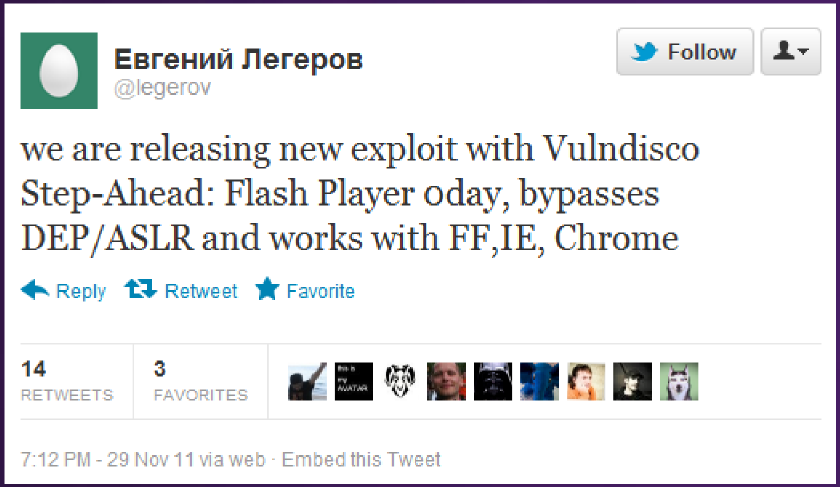
Background

- Fuzzing effort by Google researchers.

The initial run of the ongoing effort resulted in about 400 unique crash signatures, which were logged as 106 individual security bugs following Adobe's initial triage. As these bugs were resolved, many were identified as duplicates that weren't caught during the initial triage. A unique crash signature does not always indicate a unique bug. Since Adobe has access to symbols and sources, they were able to group similar crashes to perform root cause analysis reducing the actual number of changes to the code. No analysis was performed to determine how many of the identified crashes were actually exploitable. However, each crash was treated as though it were potentially exploitable and addressed by Adobe. In the final analysis, the Flash Player update Adobe shipped earlier this week contained about 80 code changes to fix these bugs.

Background

- There could be even more unknown Flash threats than what we’re aware of now…



Background

- Check out Peleus Uhley's CSW 2012 talk for more info on Flash threats from the vendor's perspective.

  http://cansecwest.com/csw12/CSW2012-AdvPersistentResponse.pdf

Background

- Most Flash vulnerabilities are ActionScript-related.
- ActionScript can be used for exploitation - not just for Flash vulns, for example:
  - AS heap spray (old-school)
  - JIT Spray (Dion Blazakis)
  - JIT type confusion (my work)
  - AS-level info-leak (see @fjserna’s recent work)
  - More to come?
- We have a “weird” machine here! *(word from Halvar Flake’s Infiltrate 2011 talk)*

Background

- What's the coding fault actually is for a specific Flash vulnerability? (we are not discussing on the AS level, but "ASM" level)
- Has Adobe patched them correctly? (From a defense point of view, we sometimes need to ensure they are patched correctly).
- Are there more effective ways to find Flash bugs?

Background

- Flash ActionScript Virtual Machine has not been studied very well.
- We want to provide better “answers” for these questions, and meet current and future challenges from Flash threats.

Methodology

- Reviewed the Tamarin Project (open-sourced AVM).
- Reverse Engineered released Flash Player.
- Developed analysis tools to help with this research and future analysis. We will introduce them.

Agenda

1	Introducing ASParser
2	AVM Implementation
3	Bytecode Verifier
4	The Fault
5	Fuzzing in Memory

Introducing ASParser

- A parser focusing on Flash ActionScript-related structures.
  - From researchers' perspective
  - Born for deep AVM test/research
  - A 010 Editor template
  - More than 4000-lines code

- The major difference from other Flash/AS tools.
  - You can locate/modify the "bytes" (so, the value) you want to change directly, working as a "WYSIWYW" way.
  - Based on 010 Editor template (open-sourced), you can extend the function easily, such as writing a 010 Editor script to targeted-fuzz the fields you want.

Introducing ASParser

Nothing more valuable than a demo. ☺

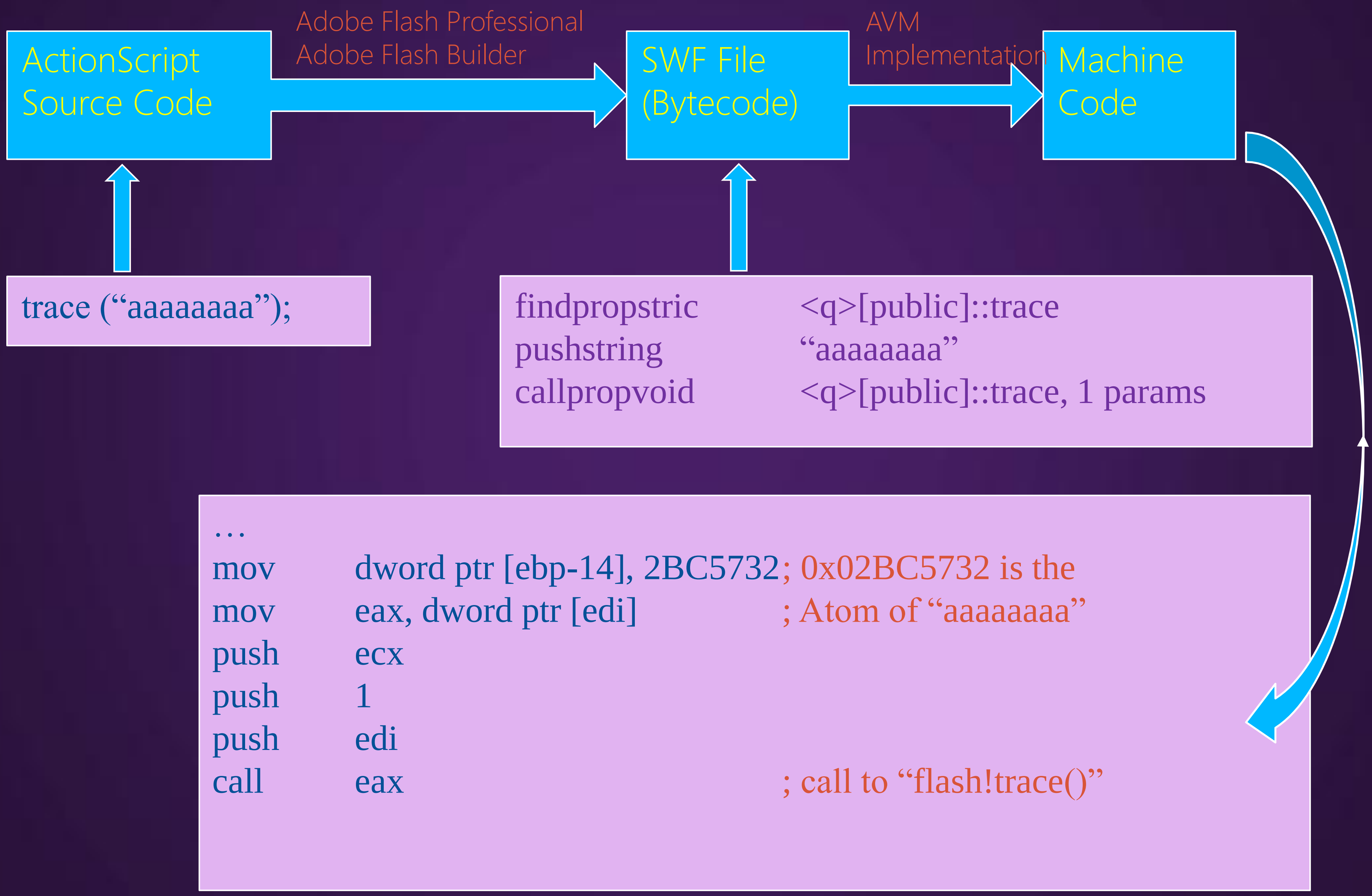
Introducing ASParser

If you are interested,
just ask me after the presentation.
☺

Agenda

1	Introducing ASParser
2	AVM Implementation
3	Bytecode Verifier
4	The Fault
5	Fuzzing in Memory

How Your AS Code Works



AVM Implementation

- Based on method (function)

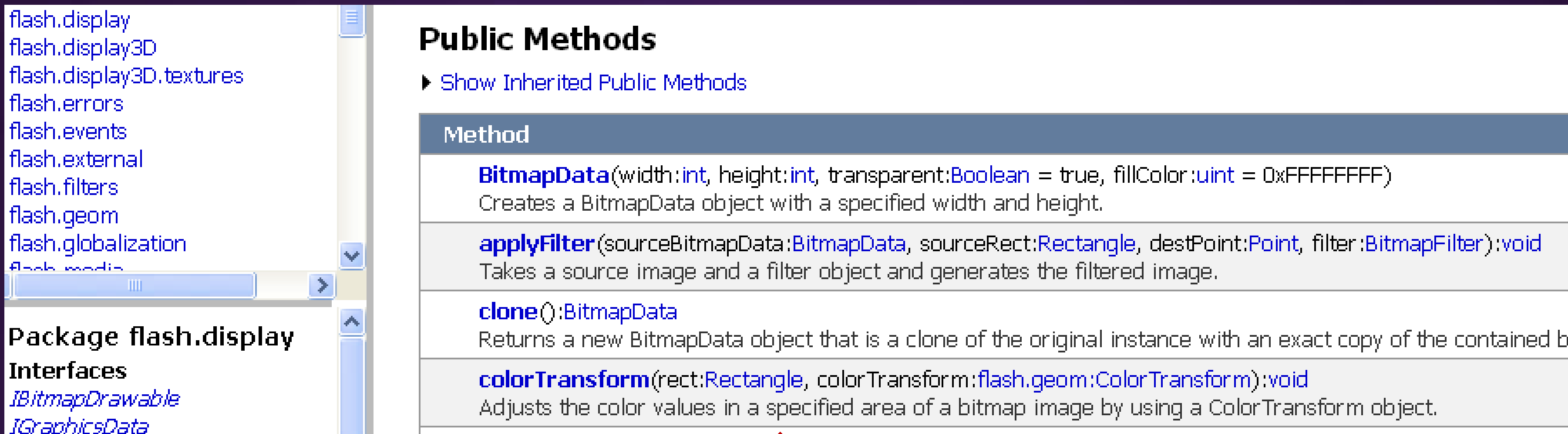
⊞ struct U30 method_body_count	00000004
⊟ struct METHOD_BODY_INFOs method_bodys	
⊞ struct METHOD_BODY_INFO method_body[0]	{class_static_init} <q>[public]::myClass extends <q>[public]flash.display::MovieClip
⊞ struct METHOD_BODY_INFO method_body[1]	{instnc,method} <q>[public]::myClass => <q>[packageinternal]::test
⊞ struct METHOD_BODY_INFO method_body[2]	{instnc,cnstrt} <q>[public]::myClass
⊞ struct METHOD_BODY_INFO method_body[3]	{script_init} <Entry> name: <q>[public]::myClass, class: <q>[public]::myClass extends

AVM Implementation

- Based on method (function)

struct U30 method_body_count	00000004
struct METHOD_BODY_INFOs method_bodys	
struct METHOD_BODY_INFO method_body[0]	{class_static_init} <q>[public]::myClass extends <q>[public]flash.display::MovieClip
struct METHOD_BODY_INFO method_body[1]	{instnc,method} <q>[public]::myClass => <q>[packageinternal]::test
struct METHOD_BODY_INFO method_body[2]	{instnc,cnstrt} <q>[public]::myClass
struct METHOD_BODY_INFO method_body[3]	{script_init} <Entry> name: <q>[public]::myClass, class: <q>[public]::myClass extends

- 3 kinds of methods
  - Native methods, no bytecode & invisible, pointing to a function in Flash Player binary.
  - Static-init methods, invisible by Flash developers, generated by Flash compliers.
  - Normal methods, the code we write in .as files.



trace("aaaa");

- 3 kinds of methods
  - Native methods, no bytecode & invisible, pointing to a function in Flash Player binary.

AVM Implementation

- Based on method (function)

⊞ struct U30 method_body_count	00000004
⊟ struct METHOD_BODY_INFOs method_bodys	
⊞ struct METHOD_BODY_INFO method_body[0]	{class_static_init} 〈q〉[public]::myClass extends 〈q〉[public]flash.display::MovieClip
⊞ struct METHOD_BODY_INFO method_body[1]	{instnc,method} 〈q〉[public]::myClass => 〈q〉[packageinternal]::test
⊞ struct METHOD_BODY_INFO method_body[2]	{instnc,cnstrt} 〈q〉[public]::myClass
⊞ struct METHOD_BODY_INFO method_body[3]	{script_init} 〈Entry〉 name: 〈q〉[public]::myClass, class: 〈q〉[public]::myClass extends

method_body[3]

method_body[0]

- 3 kinds of methods
  - Static-init methods, invisible by Flash developers, generated by Flash compliers automatically.

AVM Implementation

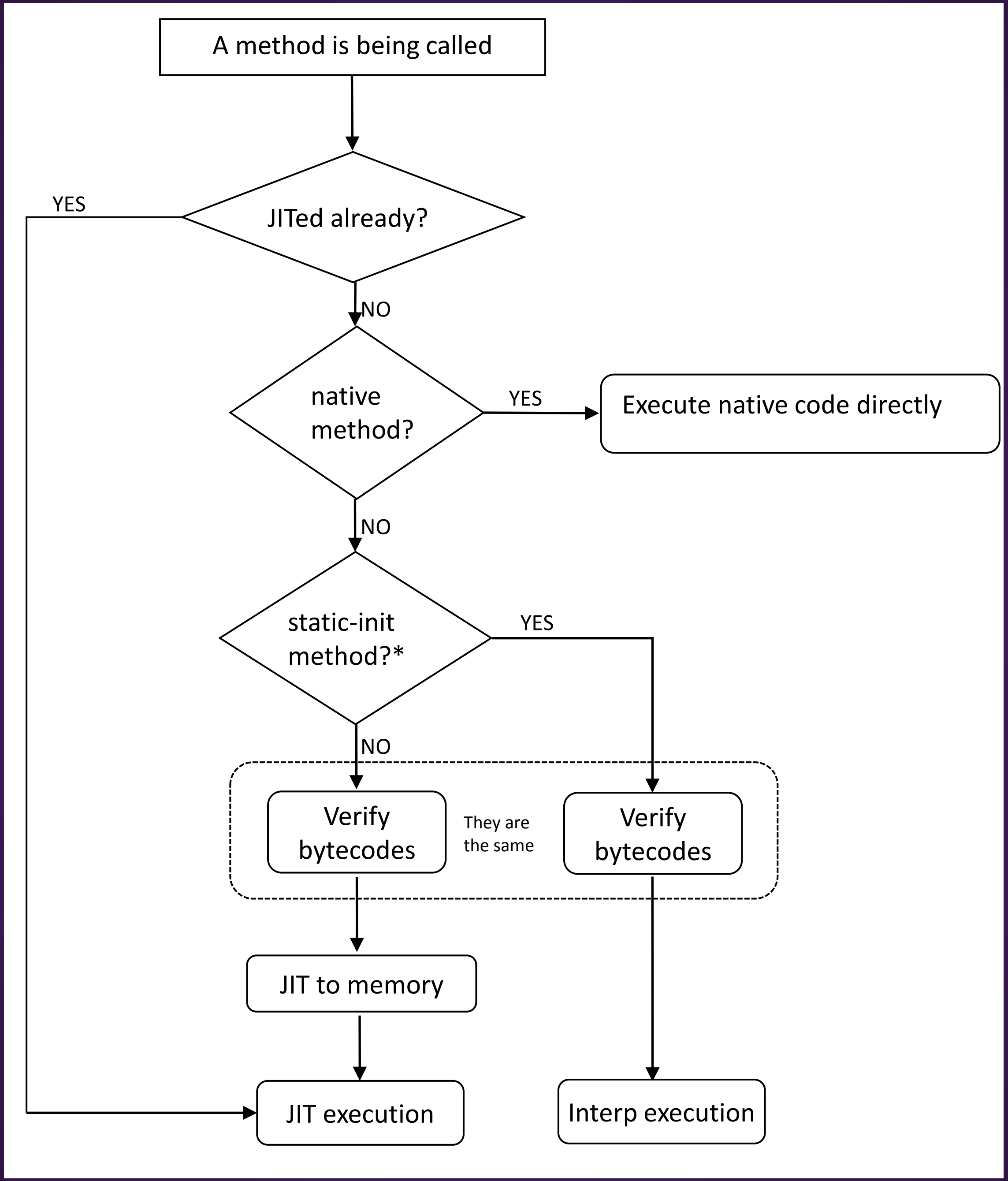
- Based on method (function)

struct U30 method_body_count	00000004
struct METHOD_BODY_INFOs method_bodys	
struct METHOD_BODY_INFO method_body[0]	{class_static_init} <q>[public]::myClass extends <q>[public]flash.display::MovieClip
struct METHOD_BODY_INFO method_body[1]	{instnc,method} <q>[public]::myClass => <q>[packageinternal]::test
struct METHOD_BODY_INFO method_body[2]	{instnc,cnstrt} <q>[public]::myClass
struct METHOD_BODY_INFO method_body[3]	{script_init} <Entry> name: <q>[public]::myClass, class: <q>[public]::myClass extends

```
package  {
    import flash.display.MovieClip;
    public class myClass extends MovieClip {
        function test() {
            var b = (0x3c54d0d9 ^ 0x3c909058);
        }

        function myClass() {
            var a = (0x41414141 ^ 0x42424242);
        }
    }
}
```

- 3 kinds of methods
  - Normal methods, the code we write in .as files.



*For some certain special situations such as the method has too many parameters, AVM will choose not to JIT but use Interp, please check BaseExecMgr::shouldJitFirst() for more info.

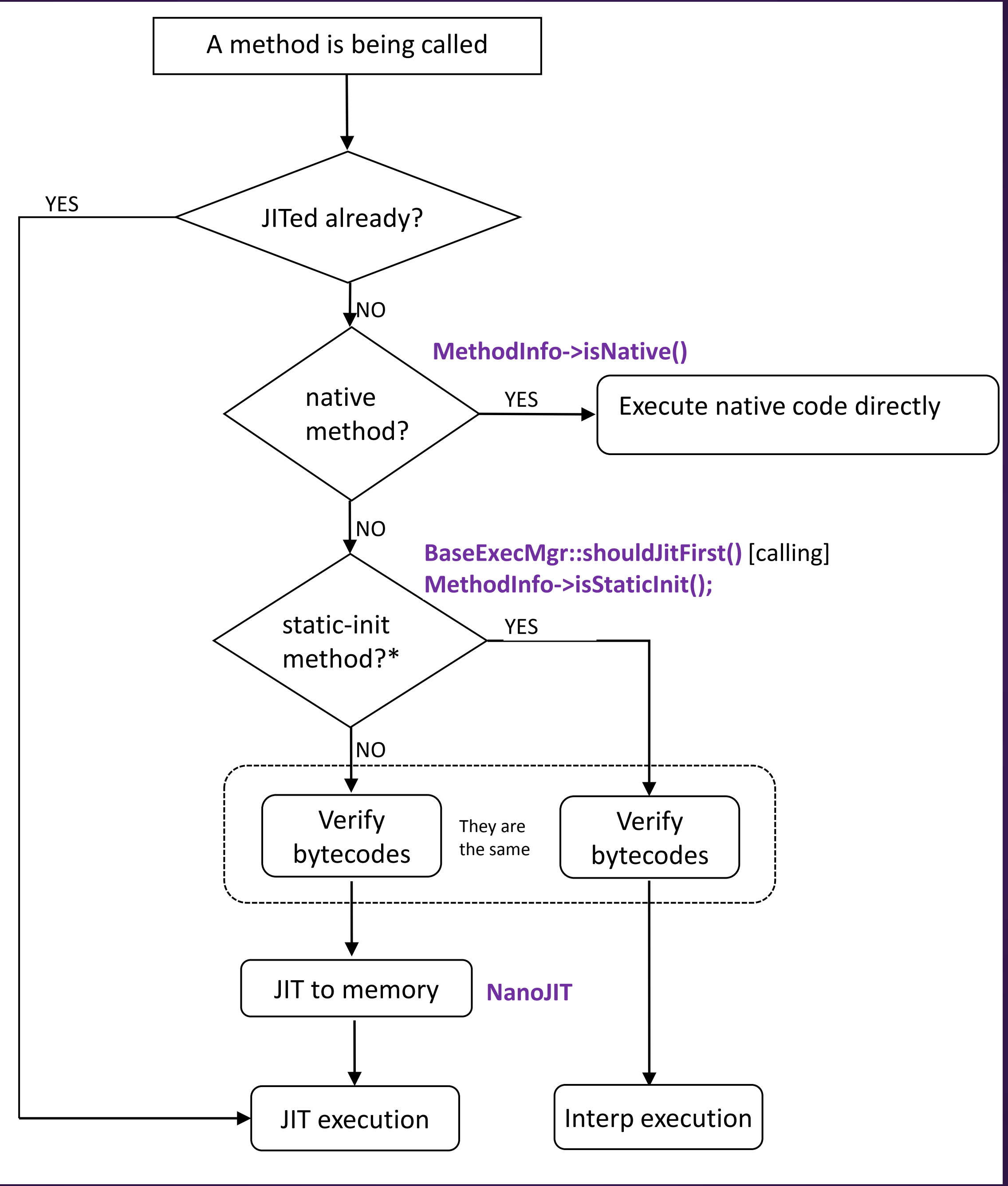
AVM Implementation

- Native methods will be called directly in .text.
- Static-init methods will be executed in interpreter mode
- **Normal methods**
  - Verified first
  - NanoJIT to generate JITed code in memory
  - Only JITed at the 1st time being called
  - If called in future, go to the JITed code directly

Normal methods are our focus in this presentation

AVM Implementation

Add comments for key functions
called during the implementation..



*For some certain special situations such as the method has too many parameters, AVM will choose not to JIT but use Interp, please check BaseExecMgr::shouldJitFirst() for more info.

MethodInfo class

MethodInfo	*stores all the info for a method*
+4h	*JITed func pointer or interp entrance* *- if the pointer is beyond the binary scope, the function is JITed*
+20h	*Method id ("method" in "method_body")* *- AVM uses this to identify a method*
+24h	*Pointer to "max_stack"(skipping "method_id")* *- use this to get the bytecode & method env*
+34h	*flags (isNative, isStaticInit, etc)* *- read the DWORD to determine what kind of method it is*

Function relations

- verifyEnterGPR/verifyEnterFPR
  - verifyOnCall
    - verifyMethod
      - verifyJit
        - verifyCommon
          - Verifier::verify()
        - setJIT
      - verifyInterp
        - verifyCommon
          - Verifier::verify()
        - setInterp

Go Real JIT or Interp Execution

What we know for a Flash/AS

- Not all the ActionScript methods will be executed when you open a Flash.
- Some compiler-generated methods (“Static-init”) will not be JITed.
- By finding the corresponding functions on a released Flash Player, we can debug ActionScript on our debugger.
  - Just like you debugging any application!

Debugging ActionScript

- Hook in AbcParser::parseMethodBodies

```
void AbcParser::parseMethodBodies()
  {
      int bodyCount = readU30(pos);
      ..
      for (int i=0; i < bodyCount; i++)
      {
          ..
          uint32_t method_index = readU30(pos);
          MethodInfo* info = resolveMethodInfo(method_index);

          const uint8_t *body_pos = pos;

          //record the id of method_body, since we need to know this in collaborating with the result of our ASParser
```

- Hook at the end of verifyOnCall
  - When method_body is JITed or Interped
  - Read method_body_pos (+24h) and JITed address (+4h) from MethodInfo

- We can trace the execution of ActionScript on Flash!

!ASDebugger Demo1 - trace

- Before loading Flash file:
  - !ASDebugger -i //install our hooks
- After loading Flash file:
  - !ASDebugger -d //display the trace info
- You can also export the “symbol” from ASParser and load it so you have more friendly output.
  - !ASDebugger -d -s c:\asTest\symbol

!ASDebugger Demo1 - trace

```
0BADF00D [trace] method_body[744]    JITed at  0x02C42F13    {class,static,method} <q>[public]fl.controls::TextArea extends <q>[public]fl.core::UICo
0BADF00D [trace] method_body[676]    JITed at  0x02C42E8C    {class,static,method} <q>[public]fl.controls::ScrollBar extends <q>[public]fl.core::UIC
0BADF00D [trace] method_body[292]    JITed at  0x02C42A50    {class,static,method} <q>[public]fl.core::UIComponent extends <q>[public]flash.display:
0BADF00D [trace] method_body[27]     JITed at  0x02C42741    {class,static,method} <q>[public]fl.managers::StyleManager extends <q>[public]::Object
0BADF00D [trace] method_body[28]     JITed at  0x02C42402    {class,static,method} <q>[public]fl.managers::StyleManager extends <q>[public]::Object
0BADF00D [trace] method_body[325]    JITed at  0x02C42123    {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]::setSharedStyle
0BADF00D [trace] method_body[19]     interp    execution     {script_init} name: <q>[public]fl.core::InvalidationType, class: <q>[public]fl.core::In
0BADF00D [trace] method_body[17]     interp    execution     {class_static_init} <q>[public]fl.core::InvalidationType extends <q>[public]::Object
0BADF00D [trace] method_body[324]    JITed at  0x02C41F63    {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]::invalidate
0BADF00D [trace] method_body[345]    JITed at  0x02C41CB0    {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core:UIComponent::c
0BADF00D [trace] method_body[789]    JITed at  0x02C4140C    {instnc,method} <q>[public]fl.controls::TextArea => <q>[protected]fl.controls:TextArea:
0BADF00D [trace] method_body[337]    JITed at  0x02C41156    {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core:UIComponent::c
0BADF00D [trace] method_body[338]    JITed at  0x02C40CCA    {instnc,method} <q>[public]fl.core::UIComponent => <q>[protected]fl.core:UIComponent::c
0BADF00D [trace] method_body[300]    JITed at  0x02C40B58    {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]::setSize
0BADF00D [trace] method_body[105]    interp    execution     {script_init} name: <q>[public]fl.events::ComponentEvent, class: <q>[public]fl.events::
0BADF00D [trace] method_body[101]    interp    execution     {class_static_init} <q>[public]fl.events::ComponentEvent extends <q>[public]flash.event
0BADF00D [trace] method_body[102]    JITed at  0x02C40A45    {instnc,cnstrt} <q>[public]fl.events::ComponentEvent
0BADF00D [trace] method_body[308]    JITed at  0x02C40864    {instnc,method} <q>[public]fl.core::UIComponent => <q>[public]::move
0BADF00D [trace] method_body[790]    JITed at  0x02C405FD    {instnc,method} <q>[public]fl.controls::TextArea => <q>[protected]fl.controls:TextArea:
0BADF00D [trace] method_body[748]    JITed at  0x02C40566    {instnc,getter} <q>[public]fl.controls::TextArea => <q>[public]::enabled
0BADF00D [trace] method_body[298]    JITed at  0x02C404E0    {instnc,getter} <q>[public]fl.core::UIComponent => <q>[public]::enabled
0BADF00D [trace] method_body[924]    JITed at  0x02C40417    {instnc,cnstrt} <q>[public]fl.controls::UIScrollBar
0BADF00D [trace] method_body[677]    JITed at  0x02C401FF    {instnc,cnstrt} <q>[public]fl.controls::ScrollBar
0BADF00D [trace] method_body[48]     interp    execution     {script_init} name: <q>[public]fl.controls::ScrollBarDirection, class: <q>[public]fl.co
0BADF00D [trace] method_body[46]     interp    execution     {class_static_init} <q>[public]fl.controls::ScrollBarDirection extends <q>[public]::Obj
0BADF00D [trace] method_body[923]    JITed at  0x02C40086    {class,static,method} <q>[public]fl.controls::UIScrollBar extends <q>[public]fl.control
0BADF00D [trace] method_body[698]    JITed at  0x02C3F7A8    {instnc,method} <q>[public]fl.controls::ScrollBar => <q>[protected]fl.controls:ScrollBa
0BADF00D [trace] method_body[678]    JITed at  0x02C3F655    {instnc,method} <q>[public]fl.controls::ScrollBar => <q>[public]::setSize
0BADF00D [trace] method_body[863]    JITed at  0x02C3F2D0    {instnc,cnstrt} <q>[public]fl.controls::BaseButton
0BADF00D [trace] method_body[862]    JITed at  0x02C3F250    {class,static,method} <q>[public]fl.controls::BaseButton extends <q>[public]fl.core::UI
0BADF00D [trace] method_body[1]      interp    execution     {script_init} name: <q>[public]fl.managers::IFocusManagerComponent, class: <q>[public]f
0BADF00D [trace] method_body[0]      interp    execution     {class_static_init} <q>[public]fl.managers::IFocusManagerComponent
0BADF00D [trace] method_body[347]    JITed at  0x02C3F0B7    {instnc,method} <q>[public]fl.core::UIComponent => <q>[private]NULL::initializeFocusMan
0BADF00D [trace] method_body[872]    JITed at  0x02C3EE06    {instnc,method} <q>[public]fl.controls::BaseButton => <q>[protected]fl.controls:BaseBut
0BADF00D [trace] method_body[871]    JITed at  0x02C3EC9A    {instnc,method} <q>[public]fl.controls::BaseButton => <q>[public]::setMouseState
0BADF00D [trace] method_body[869]    JITed at  0x02C3EC24    {instnc,setter} <q>[public]fl.controls::BaseButton => <q>[public]::autoRepeat
0BADF00D [trace] method_body[327]    JITed at  0x02C3EB98    {instnc,setter} <q>[public]fl.core::UIComponent => <q>[public]::focusEnabled
```

```
!ASDebugger -d -s C:\asTest\symbol
```

Done

!ASDebugger Demo2 - bp

- Before loading Flash file:
  - !ASDebugger -i -b method_body_id

- Will set a breakpoint at the beginning of the JITed function, during the AVM implementation.
  - Allow debugging the JITed code for the bytecode of a method_body quickly.
  - This feature is useful since JITed memory is made “random”.
    - 2 significant mitigation features enabled from November 2011 (called “Codebase Alignment Randomization" and "Instruction Alignment Randomization“)
    - To prevent “JIT Spraying” attacks
    - Out of the scope of this project, but it’s interesting to let anyone know

!ASDebugger Demo2 - bp

```
0BADF00D  placeHardHook_AbcParserLoop on 0x008768c0
0BADF00D  TABLE SIZE: 1
0BADF00D  Logging at 0x029d0000
0BADF00D  idx = 00000005
0BADF00D  Placed hook on 00876A68
0BADF00D  p_method_body_skip_methodId=023641CF
0BADF00D  Logging on JITCall 0x0088bf6a
0BADF00D  TABLE SIZE: 1
0BADF00D  Logging at 0x02ad0000
0BADF00D  idx = 00000006
0BADF00D  fast.filterPos = 02AD0094.
0BADF00D  Placed hook on 02AD0094
7C8106F9  New thread with ID 00000358 created
0BADF00D  p_JITed_func=02C4F627, setting breakpoint on this address.
02C4F627  [13:08:38] Breakpoint at 02C4F627
!ASDebugger -i -b 678
[13:08:38] Breakpoint at 02C4F627
```

Agenda

1	Introducing ASParser
2	AVM Implementation
3	Bytecode Verifier
4	The Fault
5	Fuzzing in Memory

Bytecode Verifier

- Verification is extremely important for a compiler / interpreter (a.k.a. “virtual machine”).
- Faults on verification cause highly-dangerous JIT type confusion vulnerabilities.
  - highly-dangerous means perfect exploitation: bypassing ASLR+DEP, with %100 reliability, no heapSpray, no JITSpray.
  - JIT type confusion bugs are due to faults in the verification of AVM!
- But, what the bytecode verification exactly is?

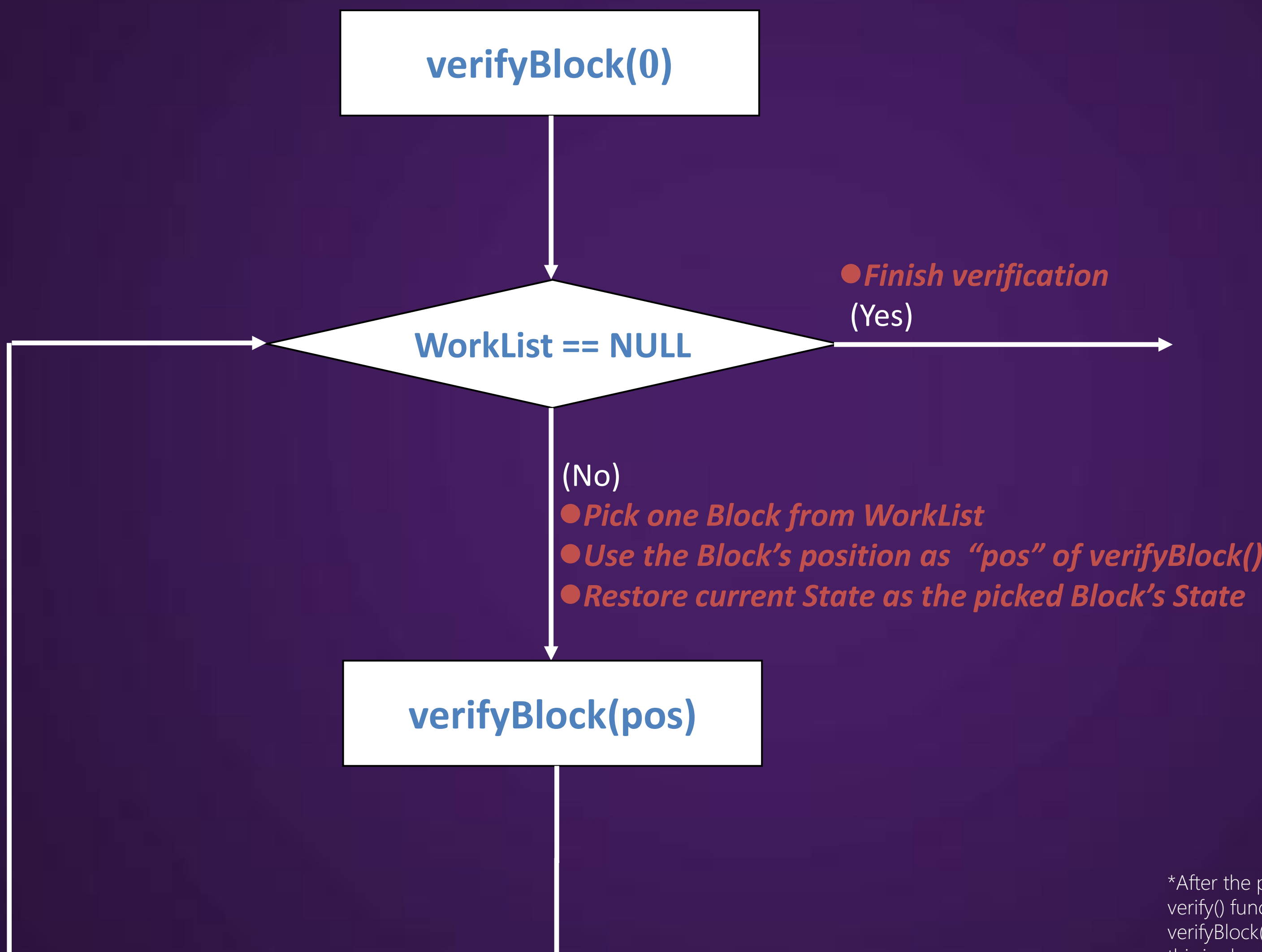
Some Concepts / Structures

- **State** - state info for a given block entry (stack, scope, stack params, types of the params, etc).
- **Block** – including the starting position of bytecode, and the State.
- **BlockList** - a list saving all the detected Blocks.
- **WorkList** - a linked-list for saving pending-verifying Blocks - the last added Block will be verified first (last detected, first verified)

Some Functions

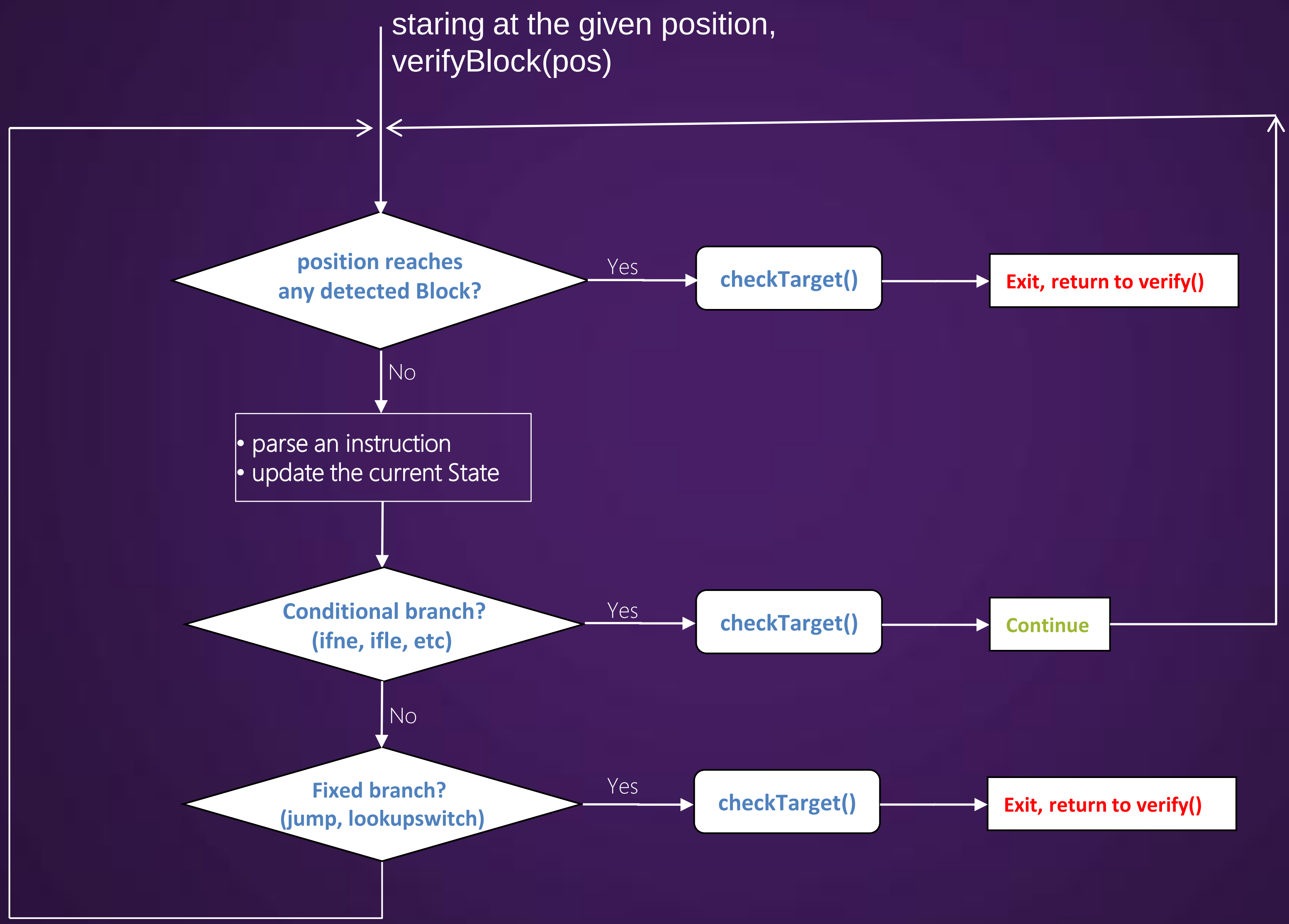
- verify() - main function
- verifyBlock(pos) - verify bytecode of a Block from pos, parse instructions.
- checkTarget(current, target) - verify the target (as indicated by the 2nd parameter)

verify()*

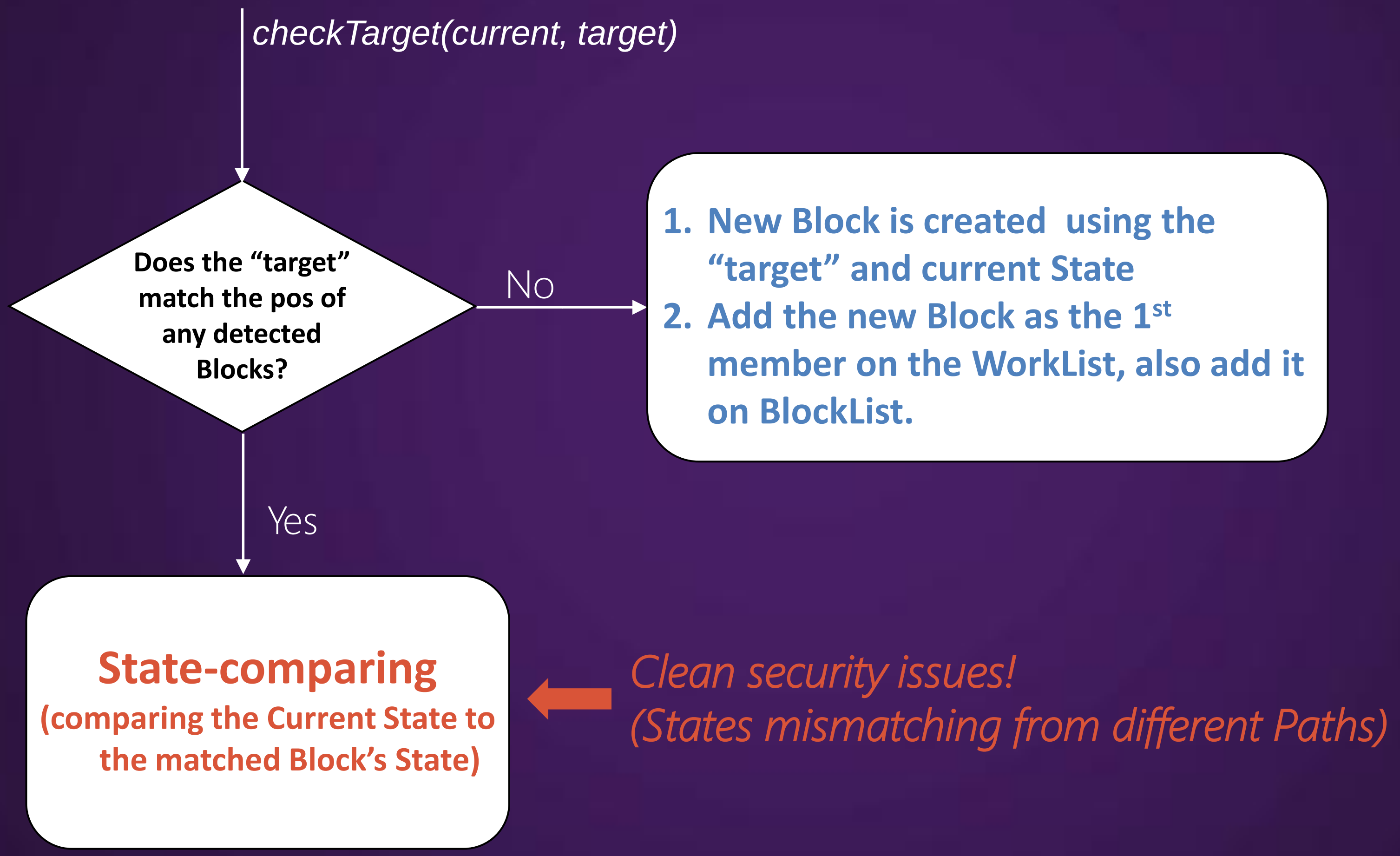


*After the process discussed in this slide, the verify() function uses another loop calling verifyBlock() to avoid overlapping instructions, this is also an important process of the verification, while it's not the focus of this research. Please check Verifier::verify() for more details.

verifyBlock()



checkTarget()



Let’s understand the logic better with an example

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
..

ins[20]:  getproperty ::length
```

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
..

ins[20]:  getproperty ::length
```

create a ByteArray object and push it on the stack.

stack[0] =
A ByteArray object

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
..

ins[20]:  getproperty ::length
```

1. Find a conditional jump, call checkTarget(12,20)
2. The target is at ins[20], and it's not in BlockList (BlockList is empty), create a new Block (Block[20]), save the current State (stack[0] = a ByteArray object) to this Block, add the new Block on BlockList/WorkList.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
..

ins[20]:  getproperty ::length
```

Continue verifying, since it’s conditional jump.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
..

ins[20]:  getproperty ::length
```

pop out the ByteArray object from the stack, push a String object ("aaaa") instead.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 p
ins[6]:   coerce <q>flash.utils
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 1
ins[12]:  ifne -> 20
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
..

ins[20]:  getproperty ::length
```

1. For every ins, it checks if the current position reaches in any detected Blocks *(we skipped this step for above ins)*.
2. At this point (ins[20]), it does match, so it call *checkTarget()*
3. *checkTarget()* will compare the current State with the State of Block[20] which was previously detected. *Though type of stack[0] doesn't match (String vs ByteArray),*

security issue is cleaned.

Why?

Recognizing State-mismatching

- If this is a serious mismatching, verification throws error.
  - AVM implementation terminated, no real execution, no bug.

- If the mismatching can be resolved, additional machine code will be generated during the JIT process, to do additional checking & etc.
  - Mismatching will be resolved in the runtime, no bug.

Thoughts

- For most situations the logic works well. Since in a straight-forward verification flow, new targets will be detected first.

When the BLUE verification flow reaches this ORANGE point, the RED path has been detected already (it will cause State-comparing)

- But, the problem is there is a possibility that the straight-forward verification flow can be performed before that the new target is detected (the BLUE one goes before RED one).

Agenda

1	Introducing ASParser
2	AVM Implementation
3	Bytecode Verifier
4	The Fault
5	Fuzzing in Memory

Let’s see another example.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

stack[0] =
A ByteArray object

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. Detected Block[22].
2. Call checkTarget(12,22), add it on WorkList/BlockList.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

Continue verification, since it’s conditional jump.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

Stack[0] =
A String Object

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteAr
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. Detected Block[30], call *checkTarget(17,30)*
2. Since the target (ins[30]) is not a detected Block (BlockList/WorkList contain only Block[22]), create a new Block (Block[30]), save the current State (stack[0] = a "aaaa" String object) to this Block, add the new Block on BlockList & WorkList
3. Since it's a fixed jump, return to verify()

Return to main function (verify())

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. At this point, on WorkList:
   Pending[0] = Block[30]
   Pending[1] = Block[22]
   Last detected = First to be verified

2. So, it will call *verifyBlock()* again starting from Block[30]

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. Retire the State (stack[0] = a "aaaa" String object), start verifying.
2. Verifying succeed since "String" object does have the "length" property.
3. Return until "ret"

Return to main function (verify())

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. At this point, on WorkList:
   Pending[0] = Block[22]
   Block[30] has been verified so it has been picked off from the list
2. So, it will call *verifyBlock()* again starting from Block[22]

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. Do nothing but detecting Block[31], call *checkTarget(22,31)*
2. New Block (Block[31]) is detected and added to BlockList & WorkList.
3. Return to *verify()* since it's a fixed jump.

Return to main function (verify())

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. At this point, on WorkList:
   Pending[0] = Block[31]
   Though the verification of Block[22] is finished, a new Block[31] has been created during the process of verifying Block[22]

2. So, it will call *verifyBlock()* again starting from Block[31]

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

1. Retire the State (stack[0]=a ByteArray object) and start verifying.
2. Verifying succeed since "ByteArray" object has the "length" property.
3. Return until "ret".

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::Byte
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

A Summary

The Order:

1. *Block[22] detected (ins[12]).*
2. *Block[30] detected (ins[17]).*
3. *Block[30] verified.*
4. *Block[22] verified.*
5. *Block[31] detected (ins[22]).*
6. *Block[31] verified.*

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteA
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

At the Order 3

The Order:

1. *Block[22] detected.*
2. *Block[30] detected.*
3. ***Block[30] verified.***
4. *Block[22] verified.*
5. *Block[31] detected (ins[22]).*

When verifying Block[30], Block[31] hasn't been detected.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteA
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

Path B

Path A

A Summary

The Order:

1. *Block[22] detected.*
2. *Block[30] detected.*
3. *Block[30] verified.*
4. *Block[22] verified.*
5. ***Block[31] detected (ins[22]).***
6. *Block[31] verified.*

When detecting Block[31] (Path A), since the Order 3 (Path B) has been performed before, there will be no State-comparing at ins[31], which so introduces the security issue.

Block Code Generation (JIT)

- Generating machine code based on detected Blocks.

- It uses a backwards order, generating code from the last Block to the first Block (bytecode position order).

- Block code generation order:
  - Block[31], Block[30], Block[22]

- When generating Block[31], the State info is that stack[0] = a ByteArray object (from Path A), so, JITed code is generated for:

Accessing “length” property on a ByteArray object

Block Code Generation (JIT)

- Quickly check the generated code.

```
02E77EB3    mov     edx, dword ptr [eax+8]
02E77EB6    mov     ecx, dword ptr [edx+A4]
02E77EBC    lea     edx, dword ptr [ebp-430]
02E77EC2    mov     dword ptr [ebp-430], eax
02E77EC8    mov     eax, dword ptr [ecx]
02E77ECA    push    edx
02E77ECB    push    0
02E77ECD    push    ecx
02E77ECE    call    eax
```

0xA4 is the offset for the “length” property on a ByteArray object.

It’s generating code for..

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

Path A

But the real Execution is..

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteArray
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring “aaaa”
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

won’t jump (0==0)

Path B

The Vulnerability

- CVE-2011-0609
- Bytecode verification fault, presented as a type confusion bug on JIT-level.
- VUPEN researcher Nicolas Joly and I developed "perfect" exploit independently in March, 2011, using my CSW11 method.
- This specific vulnerability was also studied in another MMPC presentation from a bytecode-level point of view.

The Fix

- Adding logic in *checkTarget()*

- When a new target (Block) is detected, if:
  - We can find that the position of a known Block is between the position of the branch and the position of the target,
  - Plus, that known Block is verified already.

- Then, we mark that the found Block is un-verified (putting it on the WorkList), which makes verifying the Block again.

```
ins[0]:   getlocal_0
ins[1]:   pushscope
..
ins[4]:   getlocal_0
ins[5]:   callproperty ::bla, 0 params
ins[6]:   coerce <q>flash.utils::ByteA
..
ins[10]:  pushbyte 0
ins[11]:  pushbyte 0
ins[12]:  ifne -> 22
..
ins[15]:  pop
ins[16]:  pushstring "aaaa"
ins[17]:  jump -> 30
..
ins[22]:  jump -> 31
..
ins[30]:  nop
ins[31]:  nop
ins[32]:  getproperty ::length
```

Path A

Path B

Path (added)

The Fixed Order

1. *Block[22] detected.*
2. *Block[30] detected.*
3. *Block[30] verified.*
4. *Block[22] verified*
5. *Block[31] detected (ins[22])*
6. *Block[31] verified.*
7. ***Block[30] re-verified (added)***

- [Order 5] When detecting Block[31] , Block[30] is that Block we found (pos_22 < pos_30 < pos_31), so, Block[30] will be re-verified.
- [Order 7] is the added re-verification, it will cause State-comparing at ins[31], to clean the security issue.

The Fix In Tamarin

- Change submitted in September, 2011 (while Adobe fixed it in Flash Player in March 2011)

tamarin-redux - changeset - 6570:3fdfa69a7573

summary | shortlog | changelog | graph | tags | files | changeset | raw | bz2 | zip | gz

Bug 640693 - Verifier bug can crash Player

author	Ruchi Lohani<rulohani@adobe.com>
	Fri Sep 09 14:43:19 2011 -0700 (at Fri Sep 09 14:43:19 2011 -0700)
changeset 6570	3fdfa69a7573
parent 6569	e9690dfd5e98
child 6571	1c4171eccc49
pushlog:	3fdfa69a7573

Bug 640693 - Verifier bug can crash Player

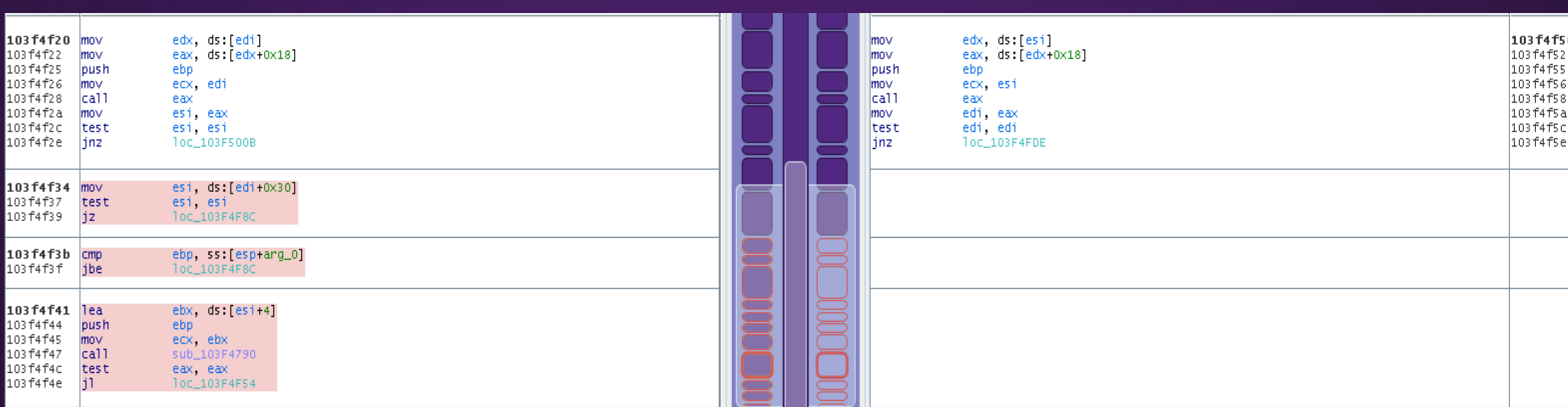
core/Verifier.cpp file | annotate | diff | revisions

```
+           if (blockStates != NULL && target > current)
+           {
+               // If we're jumping forward to an instruction with no FrameState,
+               // we need to re-verify the block that contains it to merge FrameStates.
+               // This is a conservative approach: roughly,
+               //
+               // if (target is new) and (branch < nearest < target) and (nearest not queued) then queue(nearest)
+               //
+               // this is suboptimal in that we may guess wrong and requeue blocks that don't need
+               // reverification, but this is theoretically harmless, just extra work.
+               //
+               int i = blockStates->map.findNear(target);
+               // i too large should be impossible, but i < 0 is possible, if the insertion point
+               // was before the first block. In that case, just ignore it, since we only
+               // need to requeue if it's possible that an existing block has already been
+               // verified once; in this case no such block could exist.
+               AvmAssert(i < blockStates->map.length());
+               // (But let's check anyway...)
+               if (i >= 0 && i < blockStates->map.length())
+               {
+                   // If the block we find in the table is in between the
+                   // position of the branch and the position of the target,
+                   // requeue. (And also, if it's already pending, don't bother.)
+                   FrameState* existingState = blockStates->map.at(i);
+                   if (current < existingState->abc_pc &&
+                       existingState->abc_pc < target &&
+                       !existingState->wl_pending)
+                   {
+                       #ifdef AVMPLUS_VERBOSE
+                       if (verbose) {
+                           core->console << "------------------------------------\n";
+                           core->console << "RE-QUEUE B" << int(existingState->abc_pc - code_pos) << ":";
+                       }
+                       #endif
+                       existingState->wl_pending = true;
+                       existingState->wl_next = worklist;
+                       worklist = existingState;
+                       // no return: we want to fall thru and create the new blockstate as well.
+                   }
+               }
+           }
```

The Fix on Flash Player

- It’s not that hard, we can also bindiff Flash Player. Let’s see what we got from bindiffing FP 10.2.153.1 (flash10o.ocx, fixed) and 10.2.152.32 (flash10n.ocx, affected).

sim	confi	EA primar	name primary	EA seconda	name secondary	algorithm	matche	basicblocks pri	basicblocks se	match	instr	instr	match	edges	edges
0.68	0.94	101F1F16	sub_101F1F16_10663	101E6BD8	sub_101E6BD8_31575	call reference matching	1	1	1	15	15	16	0	0	0
0.71	0.73	10170BFD	sub_10170BFD_6725	10170A30	sub_10170A30_28089	address sequence	2	2	2	3	5	4	1	1	1
0.84	0.85	1005C4FA	sub_1005C4FA_1902	1007561C	sub_1007561C_23759	call sequence matchin···	3	3	3	9	11	11	3	3	3
0.84	0.85	1004723D	sub_1004723D_1407	1005E484	sub_1005E484_23353	call sequence matchin···	3	3	3	9	11	11	3	3	3
0.84	0.85	100755FD	sub_100755FD_2391	1013A65E	sub_1013A65E_27017	call sequence matchin···	3	3	3	9	11	11	3	3	3
0.84	0.85	1002F2EE	sub_1002F2EE_830	10047213	sub_10047213_22775	call sequence matchin···	3	3	3	9	11	11	3	3	3
0.87	0.99	103F4ED0	sub_103F4ED0_20484	103F4F00	sub_103F4F00_41849	call reference matching	29	40	29	114	147	114	42	62	43
0.87	0.88	10379745	sub_10379745_17139	103A73D0	sub_103A73D0_40046	call sequence matchin···	3	3	3	10	11	11	3	3	3
0.93	0.94	100EECF0	sub_100EECF0_4418	100EB907	sub_100EB907_25721	address sequence	3	3	3	12	16	14	3	3	3
0.98	0.99	10413556	_write_string_0	104120ED	_write_string_0	name hash matching	8	8	8	31	33	37	12	12	12
0.98	0.99	104120CD	_write_string	1040DA94	_write_string	name hash matching	8	8	8	31	37	33	12	12	12
1.00	0.99	1048A8F0	__imp_CopyStgMed···	1048A8F0	__imp_CopyStgMedium	name hash matching	0	0	0	0	0	0	0	0	0
1.00	0.99	1048A8EC	__imp_CreateURLM···	1048A8EC	__imp_CreateURLMon···	name hash matching	0	0	0	0	0	0	0	0	0



Agenda

1	Introducing ASParser
2	AVM Implementation
3	Bytecode Verifier
4	The Fault
5	Fuzzing in Memory

Challenge on auditing AVM

- Because AVM is a complex machine…
- Considering you have a Flash sample:
  - Not all the AS bytecode executes when just opening - fuzzing at non-executed bytecode is just wasting your time.
  - Modifying a few bytes (1-4 bytes) can’t cover all the issues, as we know.
  - Modifying even a few more bytes will increase your fuzzing time significantly.
- Dumb-fuzzing is not a good choice for deeply auditing (though it has found many FP bugs before).

Fuzzing in memory

- Hook at *verify()*
  - Upon a normal (the 1st) entrance, save the state info (say, “state”, “coder”).
  - Change the bytecode
- Hook at *verifyFailed() / throwVerifyError()*
  - Since FP will call these functions to terminate the implementation when verification fails.
  - Restore the state, making a jump jumping back to *verify().*
- Hook at the end of *verify()*
  - Coming here means a bytecode stream is verified successfully. Let it run for full test (only do some logs).

Fuzzing in Memory

- If our bytecode-change fails on verification, program flow will jump back to verify() again*.
- Only successful byte-code change will go real Flash Player execution.

*If you implement this methodology, you need to free the memory allocated during the previous verification process, which can be resolved by hooking the Flash Player custom heap management. Otherwise, you may use a high-RAM fuzzing machine as alternative way to “resolve” the memory leaking.

The Advantage

- We use PE-patched FP to run the fuzzing, so it runs really fast.
- We can modify the bytecode stream significantly, since if error occurs, program flow will come back quickly (running in memory).
- You don’t really need many Flash samples.

The Advantage

- We fuzz during the runtime so we only fuzz on real executed bytecode streams.

- Significantly-modifying may change the method_body execution flow (so we are able to cover more rare situations), however, we will also be able to fuzz the next method_body!

mBody[a] → *mBody[a]*
mBody[b] => *mBody[**x**]*
mBody[c] → *mBody[**y**]*

Changing the flow like Chain Reaction!

Conclusion

- This research aims to help people understand ActionScript Virtual Machine better.
- It also provides necessary information on understanding AVM-based vulnerabilities thoroughly and deeply, including finding out the “coding fault”.
- By understanding AVM, not only we can debug ActionScript on our debugger, but also we can develop effective approach to audit the robustness of AVM.

Acknowledgments

- The author would like to thank the following individuals for their help and support in preparing of the presentation:

    Jeremy Dallman
    Peter Ferrie
    Tanmay Ganacharya
    Heather Goudey
    Eddy Hsia
    Gilou Tenebro
    Jeff Williams
    Peleus Uhley (Adobe)

- Thanks to *Nico Waisman* (Immunity Inc) for the help in resolving a specific problem encountered in the Immunity Debugger PyCommand development.