

v1.0.0



VPAND.COM

Virtual Processor Infrastructure for Code Reverse Engineer and VMProtect.

Loading very hard for Qt.wasm, wait a second please...



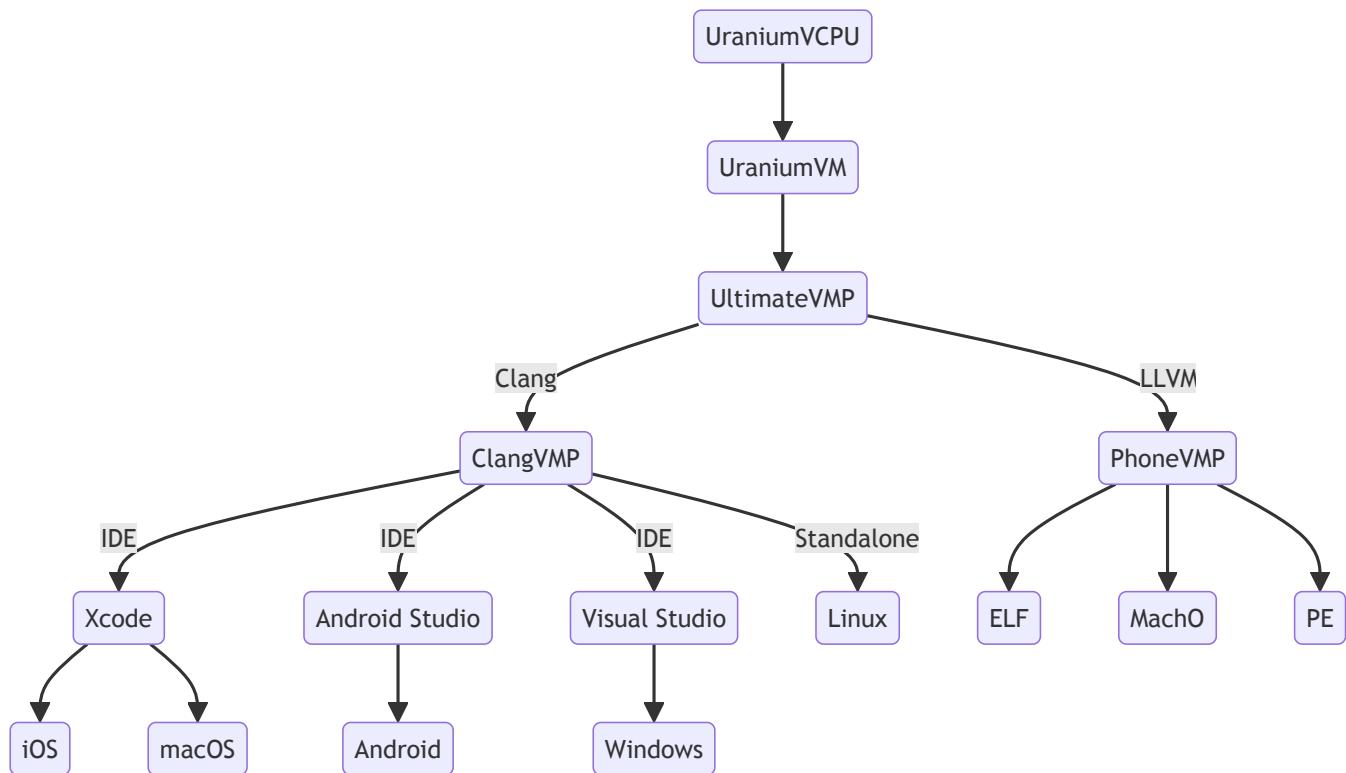
Powered by LLVM/Clang.

ClangVMP User Manual

UltimateVMP

UltimateVMP (Ultimate Virtual Machine Protection), is an **arm/arm64/x86/x86_64** assembly level code virtualization encryption software for Darwin (**macOS/iOS**), Linux (Ubuntu/**Android**), Windows and other operating systems. It re-encodes the binary instructions of the target function into a private instruction format and then interprets the encoded instruction directly at run time. Unlike low-intensity code encryption such as obfuscation and shell, VMP code will not restore the original instruction throughout the whole execution process, so it can achieve high code encryption strength, greatly raising the threshold of reverse engineering, so as to achieve the purpose of protecting software assets. UltimateVMP currently supports platforms including **macOS**, **iOS**, **Linux**, **Android**, **Windows**, and supported architectures including **x86**, **x86_64**, **arm**, and **arm64**.

UltimateVMP includes two products. One is **ClangVMP**(based on the [Clang](#) compiler), it encodes **C/C++/ObjC/ObjC++/Swift/etc.**(static native programming language) function into VMP data during compilation process. The other is **PhoneVMP**, it encodes assembly instructions into VMP data from binary file(**like MachO/ELF/PE**). All of them are based on virtual processor infrastructure [UraniumVM](#) from [vpand.com](#). Their relationship is as following:



This manual focuses on **ClangVMP**.

ClangVMP

ClangVMP is a [Clang](#)/[LLVM](#) based code virtualized compiler. It encodes **C/C++/ObjC/ObjC++/Swift/etc.**(static native programming language) function into its private VMP data during compilation process. The

VMP data will be directly interpreted by ClangVMP runtime library without restoring the original instructions. Because of this mechanism, reverse engineering on this kind of code will be much more harder than raw machine code.

Till now, ClangVMP supports running on **macOS, Windows and Linux**, and supported virtualized architectures including **x86_64 and arm64**. The following is a sample for version output on macOS installed to Xcode toolchain directory:

```
ClangVMP@vpand.com bin % ls -l
total 352320
lrwxr-xr-x 1 vpand wheel      9 Apr  8 08:22 clang -> clang-vmp
lrwxr-xr-x 1 vpand wheel      9 Apr  8 08:22 clang++ -> clang-vmp
lrwxr-xr-x 1 vpand wheel     13 Apr  8 08:22 clang++-org -> clang-
vmp-org
lrwxr-xr-x 1 vpand wheel     13 Apr  8 08:22 clang-org -> clang-vmp-
org
-rwxr-xr-x 1 vpand wheel 82115424 Apr  7 22:19 clang-vmp
-rwxr-xr-x 1 vpand wheel 98264160 Apr  7 11:47 clang-vmp-org
-rw-r--r-- 1 vpand wheel      6 Apr  7 22:19 version.txt

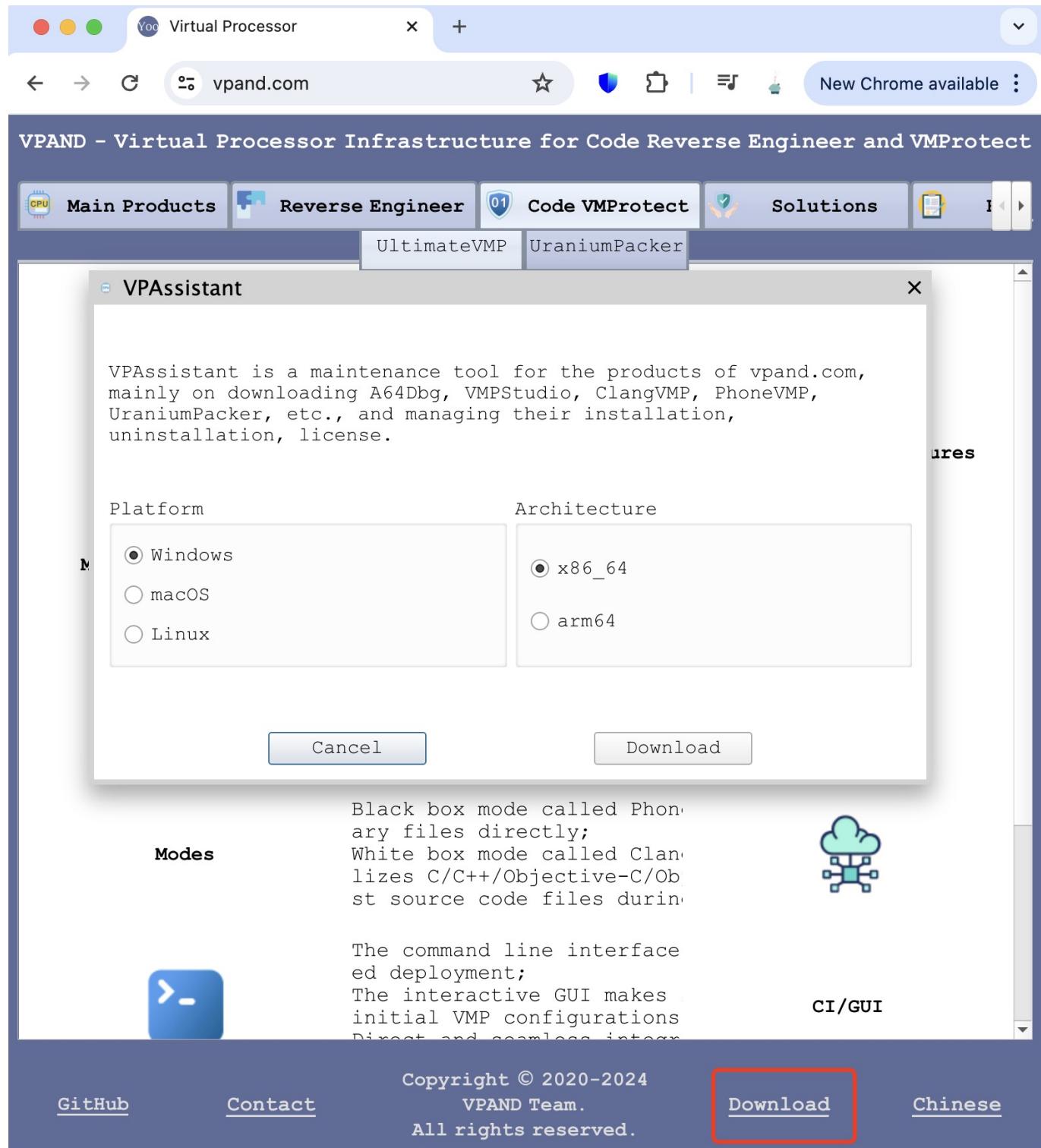
ClangVMP@vpand.com bin % ./clang-vmp --version
ClangVMP v1.1.0 Apr  7 2024 22:15:50, A LLVM/Clang(19.0.0git) based
VMProtect Compiler.
Powered by http://yunyoo.cn/ , https://vpand.com/ .

clang version 19.0.0git (https://github.com/apple/llvm-project.git)
Target: arm64-apple-darwin23.4.0
Thread model: posix
InstalledDir:
/Applications/Xcode.app/Contents/Developer/Toolchains/ClangVMP.xctoolchain
/usr/bin
```

Download

All of our products are hosted in vpand.com, you can download the assistant tool called **VPAssistant** to fetch your interested product like ClangVMP.

Please note that you should download **the exact platform and architecture** which matches your current running OS, all the real product download through VPAssistant highly depends on the architecture that it's running on. For the native performance, you'd better not download x86_64 version on arm64 macOS even though it can directly run on it with Rosetta 2.



Install

As the VPAssistant on different platform (like Windows, Linux and macOS) is **absolutely the same**, unless some feature is just available on that specified platform, otherwise all the generic feature screenshots are from macOS. Here we go.

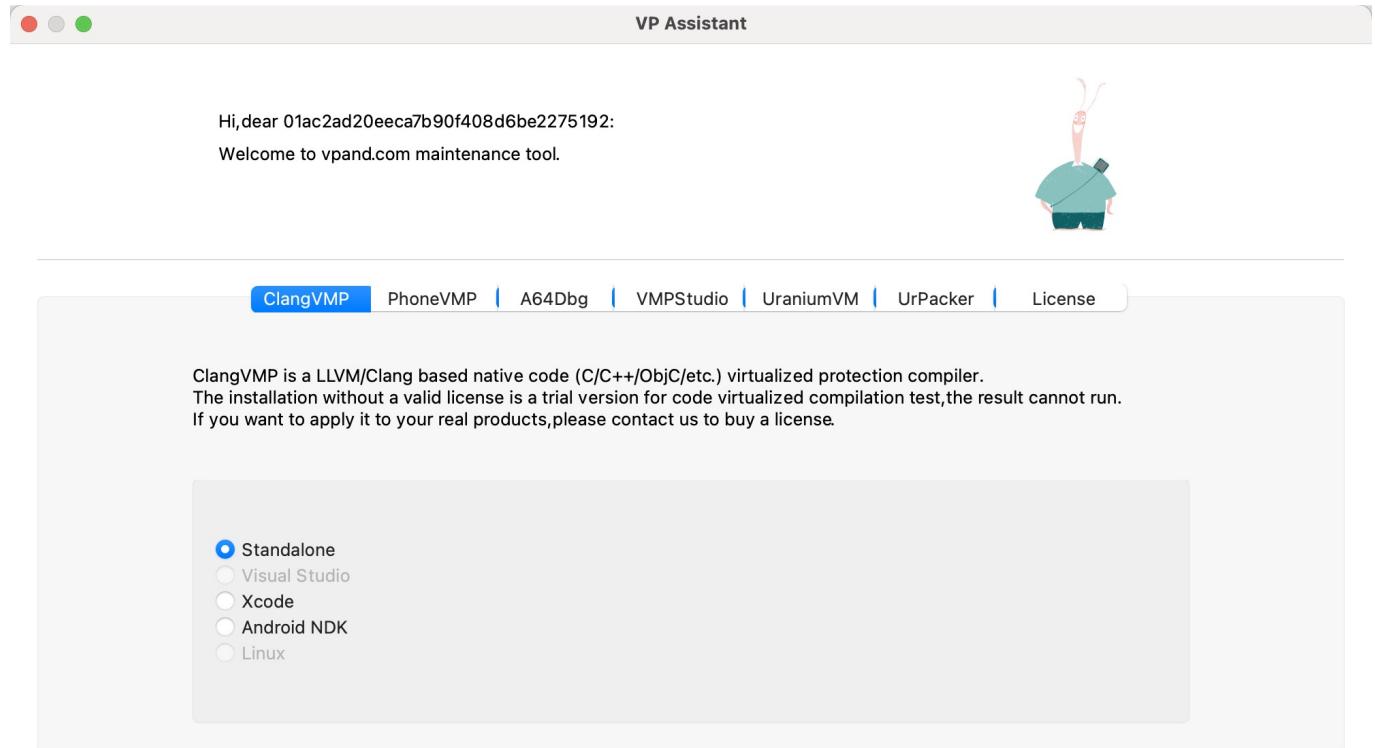
After download, unzip and launch VPAssistant, we're gonna be in the **ClangVMP** tab widget. The default installation path is (You can **change it with the "..." button** on the right):

```
macOS/Linux : ~/VPAssistant/product
```

Windows : SysDrive:\Users\user-name\VPAssistant\product

Every time when VPAssistant finishes launching, at the end of logs there'll be a line for your running OS triple: os-arch-hwid. It's the key to authenticate your computer to fetch a valid ClangVMP license, copy and send to us before you want to purchase and install a license.

current host hwid: mac-arm64-01ac2ad20eeca7b90f408d6be2275192



Installation Path: /Users/geekneo/VPAssistant/product ,0.0.0

Install

Uninstall

Update

Cancel

...

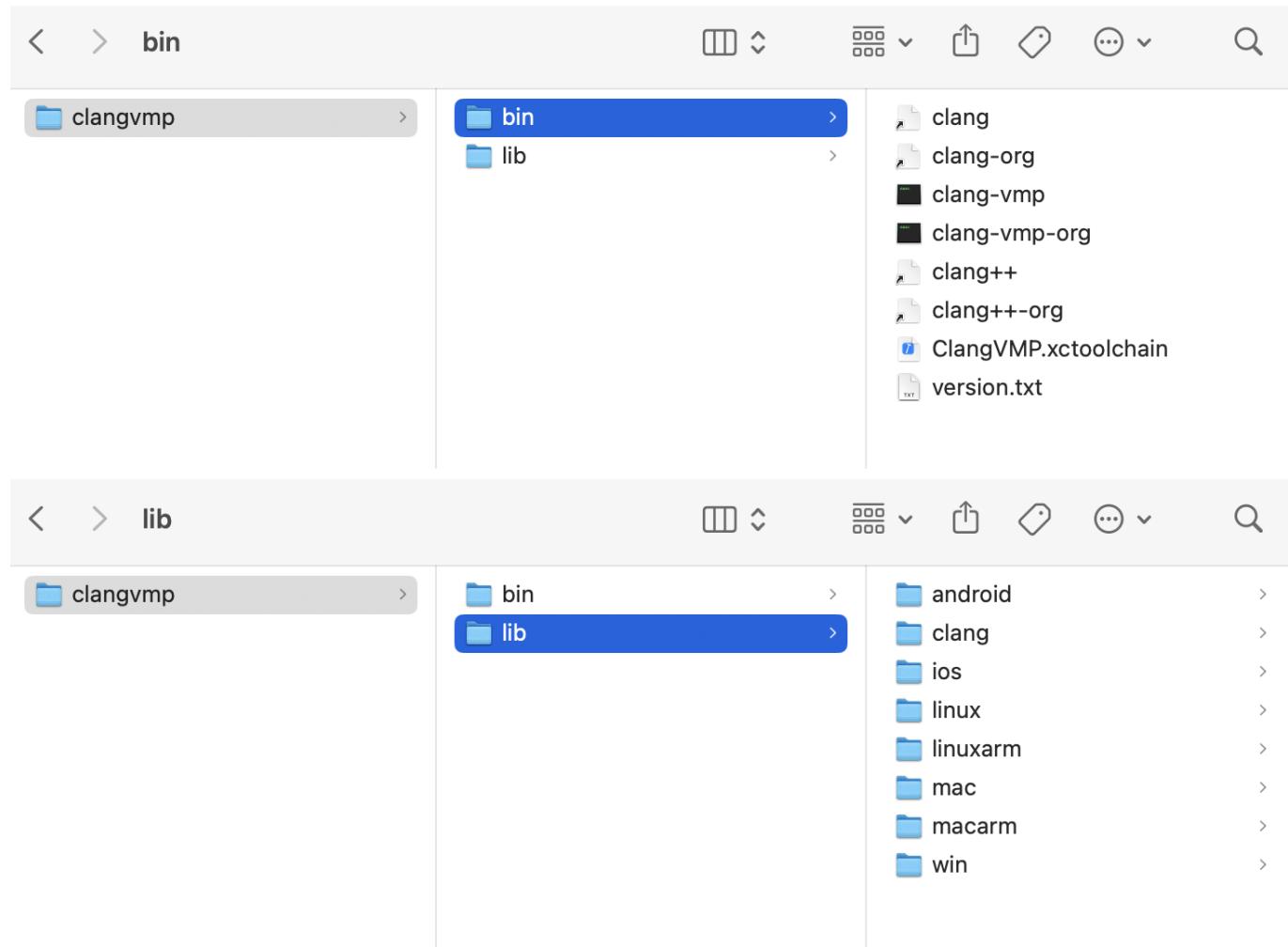
...

```
22:16:01 I : Start downloading list.txt...
22:16:04 I : Finished dowloading list.txt.
22:16:04 I : A64Dbg remote/local version is 1.17.0/0.0.0, please install it.
22:16:04 I : ClangVMP remote/local version is 1.1.0/0.0.0, please install it.
22:16:04 I : PhoneVMP remote/local version is 1.0.0/0.0.0, please install it.
22:16:04 I : UraniumVM remote/local version is 20240405.1/0.0.0, please install it.
22:16:04 I : UrPacker remote/local version is 1.0.0/0.0.0, please install it.
22:16:04 I : VMPStudio remote/local version is 1.10.0/0.0.0, please install it.
22:16:04 I : VP Assistant v1.1.1 (Apr 12 2024), current host hwid: mac-arm64-01ac2ad20eeca7b90f408d6be2275192 .
```

Standalone

The standalone installation type is suitable for any platform, it's usually for user who likes **makefile**, **ninja** or **cmake** CLI build system. After installation, it's nothing special as the generic clang compiler but needs a clangvmp.json(The VMP configuration section has detailed description) configuration file to apply its virtualized compilation feature.

Its bin and lib directory are as following(Windows and linux are nearly the same):



Visual Studio

The Visual Studio installation type will install ClangVMP as a llvm toolchain for its project solution file. Its bin directory is as following:

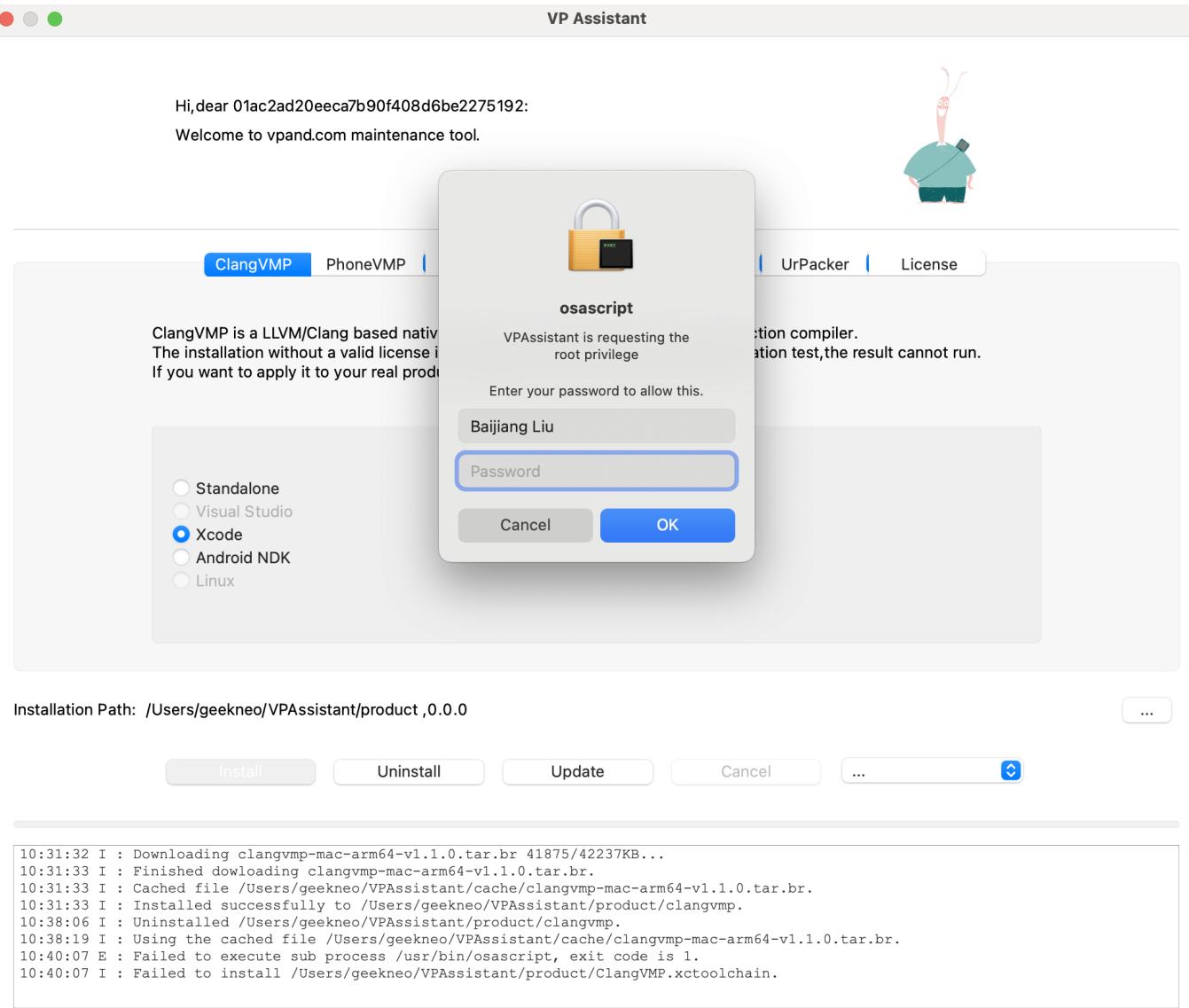
Name	Date modified	Type	Size
clang.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang++.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang++-org.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang-cl.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang-cl-org.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang-org.exe	4/19/2024 5:48 PM	.symlink	0 KB
clang-vmp.exe	4/14/2024 12:10 PM	Application	75,882 KB
clang-vmp-org.exe	3/15/2024 8:52 PM	Application	80,593 KB
Directory.build.props	4/19/2024 5:48 PM	Project Property File	1 KB
lld-link.exe	4/19/2024 5:48 PM	.symlink	0 KB
lld-link-org.exe	3/27/2024 11:12 AM	Application	45,869 KB
llvm-lib.exe	4/19/2024 5:48 PM	.symlink	0 KB
llvm-lib-org.exe	3/27/2024 11:12 AM	Application	10,455 KB
msvc140.dll	5/23/2021 4:33 PM	Application extension	661 KB
msvc140_1.dll	5/23/2021 4:33 PM	Application extension	24 KB
msvc140_2.dll	1/19/2024 8:38 PM	Application extension	263 KB
vcruntime140.dll	5/23/2021 4:33 PM	Application extension	92 KB
vcruntime140_1.dll	5/23/2021 4:33 PM	Application extension	36 KB
version.txt	4/14/2024 12:10 PM	Text Document	1 KB

Xcode

The Xcode installation type will install ClangVMP as a toolchain package into **Xcode.app**. Its fixed installation path is:

```
/Applications/Xcode.app/Contents/Developer/Toolchains/ClangVMP.xctoolchain
```

During the installation, you'll be confirmed to input the root password to move ClangVMP toolchain to Xcode.app internal directory.

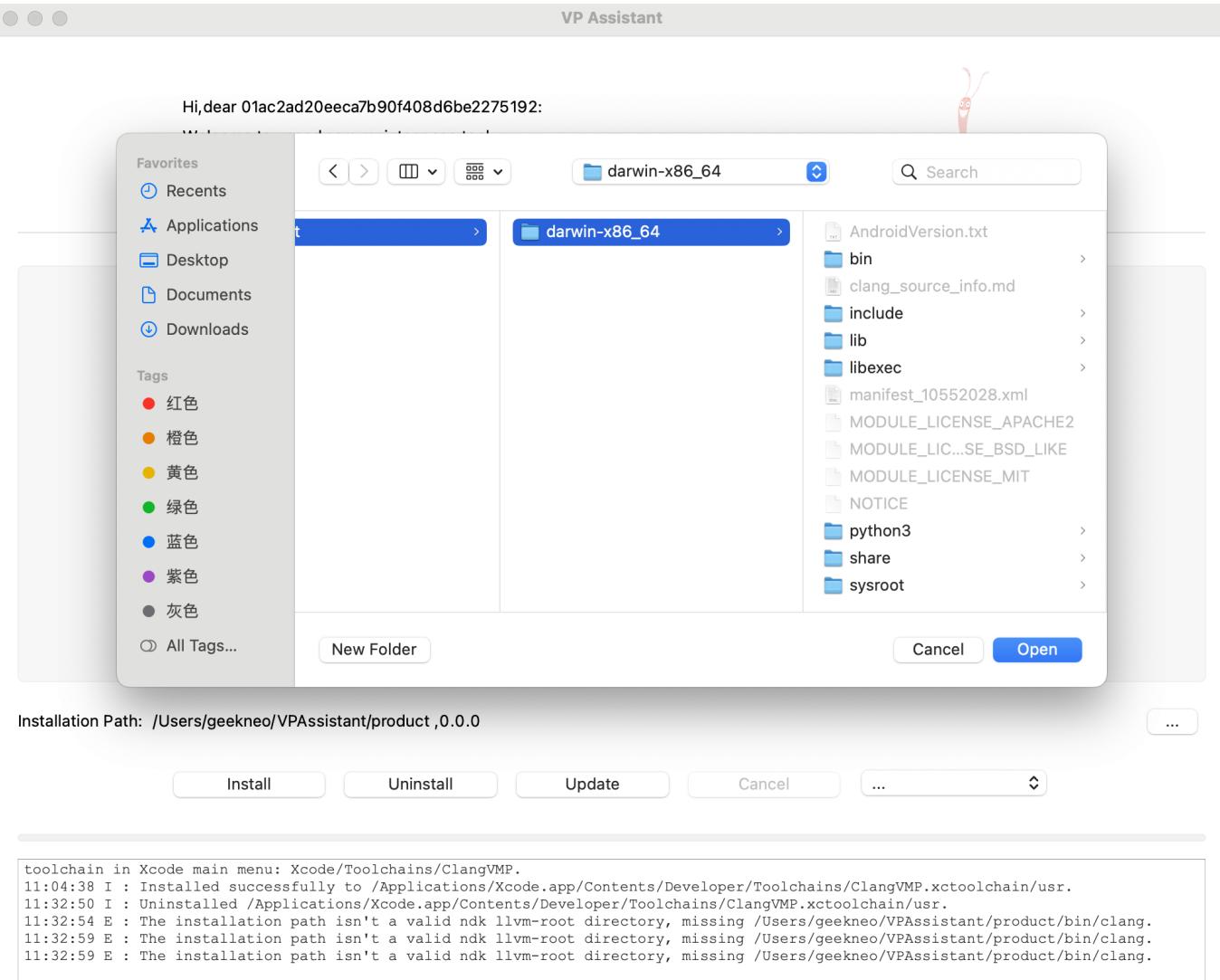


If you enter the right root password, then everything is ready for you to apply virtualized compilation in Xcode IDE.

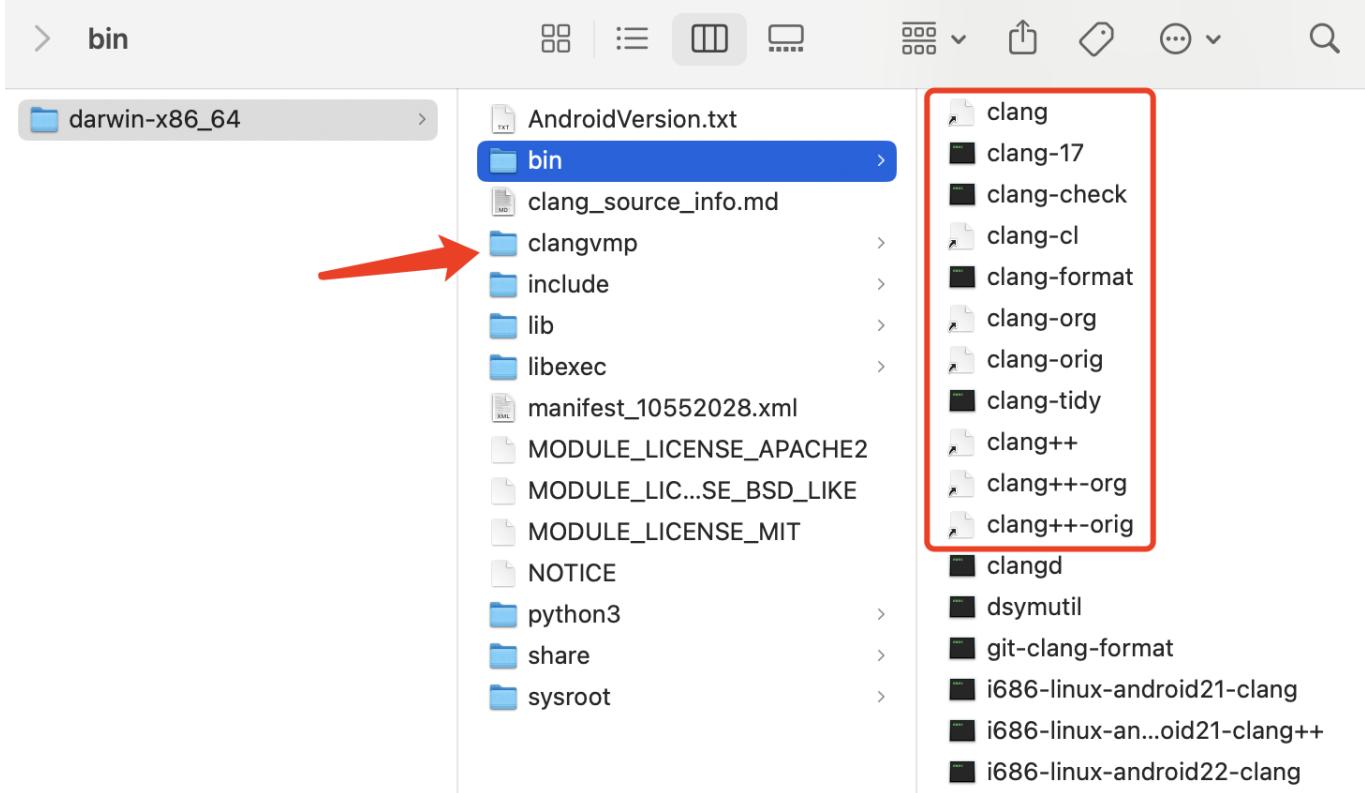
The screenshot shows the UrPacker application window. The title bar says "Toolchains". The left sidebar has categories: Applications, Library, Makefiles, Platforms, Toolchains (selected), Tools, and usr. In the main area, there's a list of toolchains: ClangVMP.xctoolchain (selected and highlighted in blue) and XcodeDefault.xctoolchain. To the right, there's a large icon of a hammer inside a shield-like shape with the word "TOOLCHAIN" below it. Below the icon, it says "ClangVMP.xctoolchain Xcode Toolchain - 196.3 MB" and "Information".

Android NDK

The Android NDK installation type will install ClangVMP into **Android NDK LLVM toolchain**. Before installation, you should select the llvm root directory of the target NDK toolchain manually.



After installation, the ndk llvm toolchain directory will be as following:

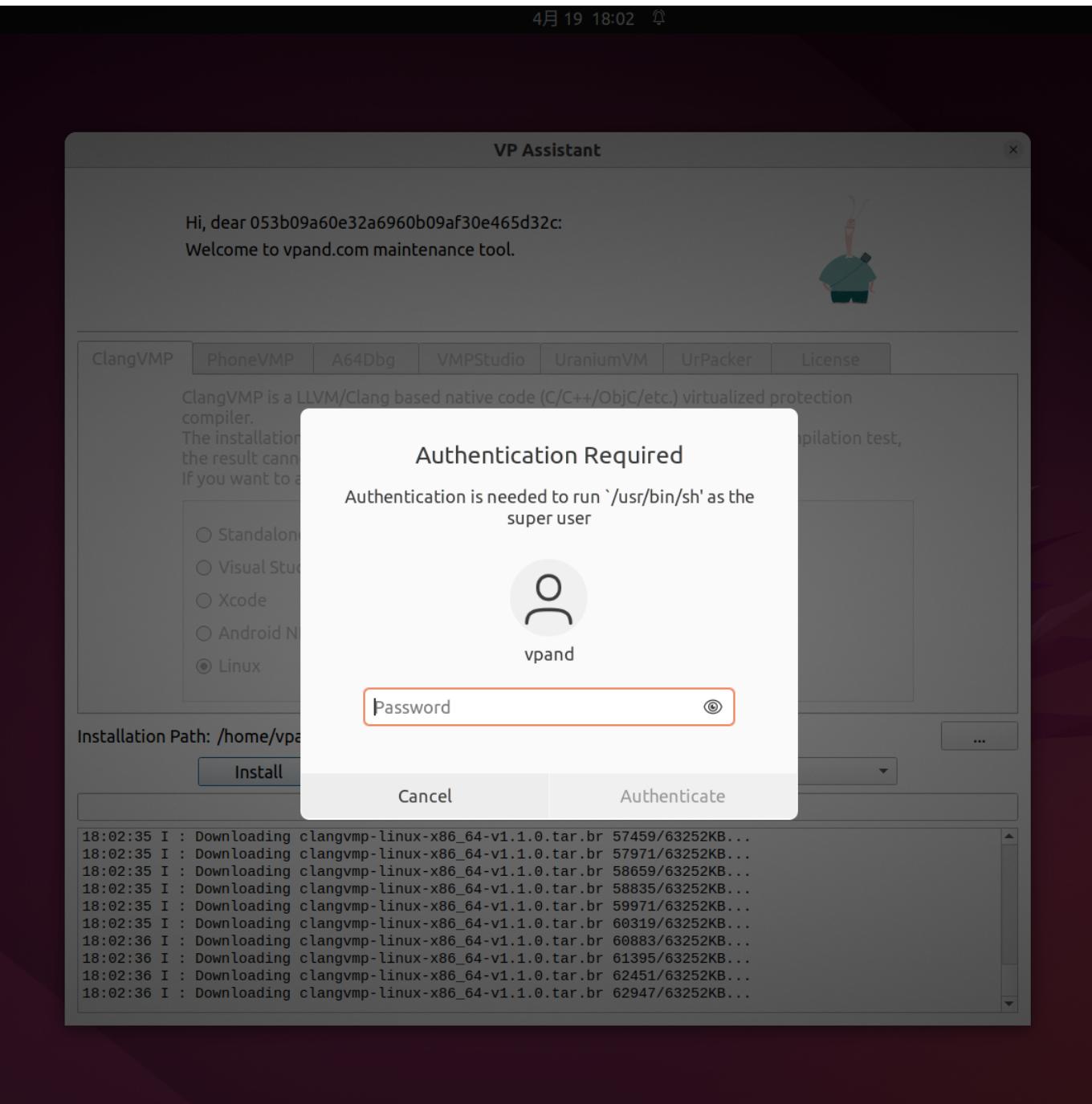


Linux

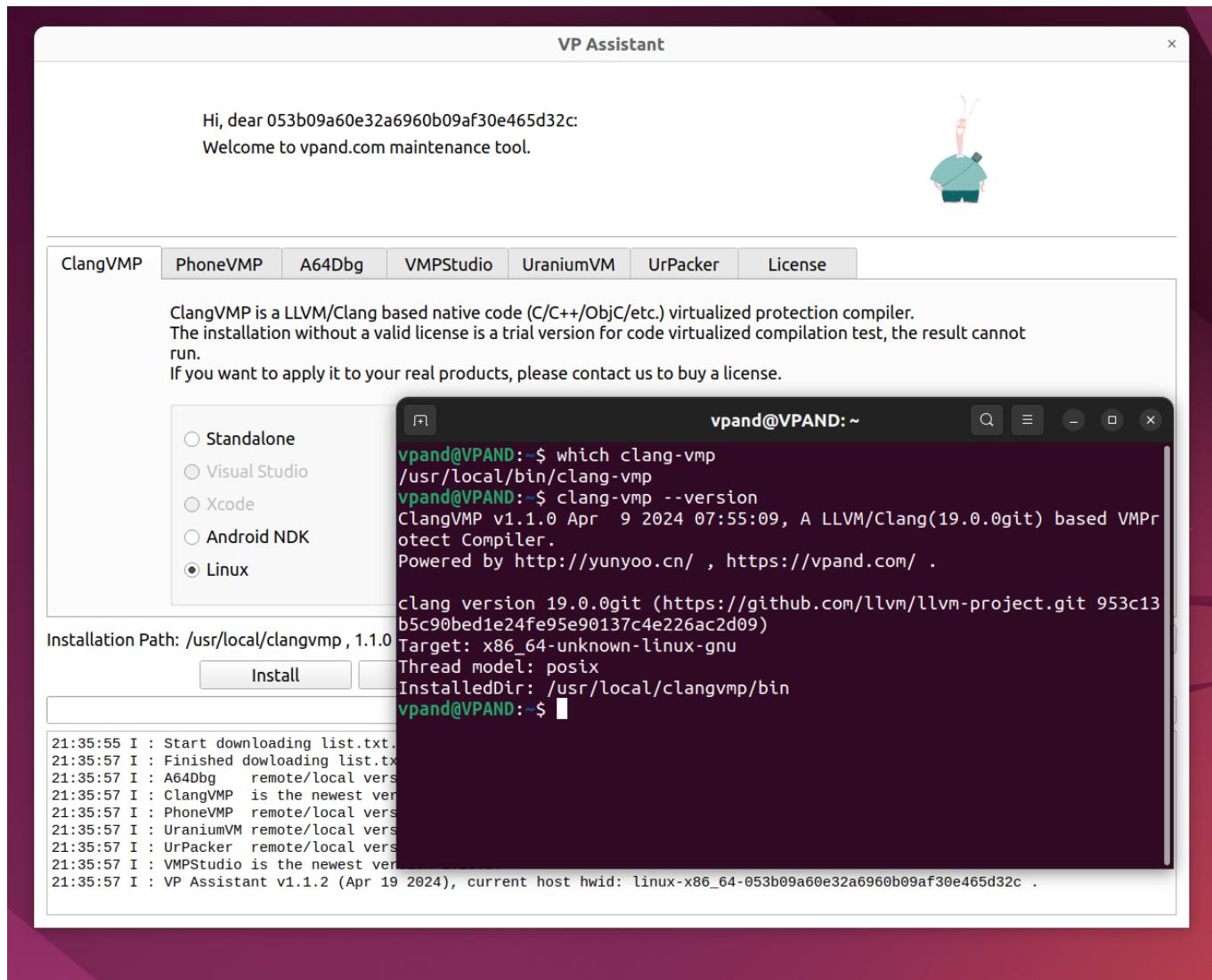
The Linux installation type will install ClangVMP as a cli tool into **/usr/local**. Its fixed installation path is:

```
/usr/local/clangvmp
```

During the installation, you'll be confirmed to input the root password to move ClangVMP toolchain to /usr/local directory.



If you enter the right root password, then everything is ready for you to apply virtualized compilation with clang-vmp cli compiler.



IDE Configuration

Visual Studio

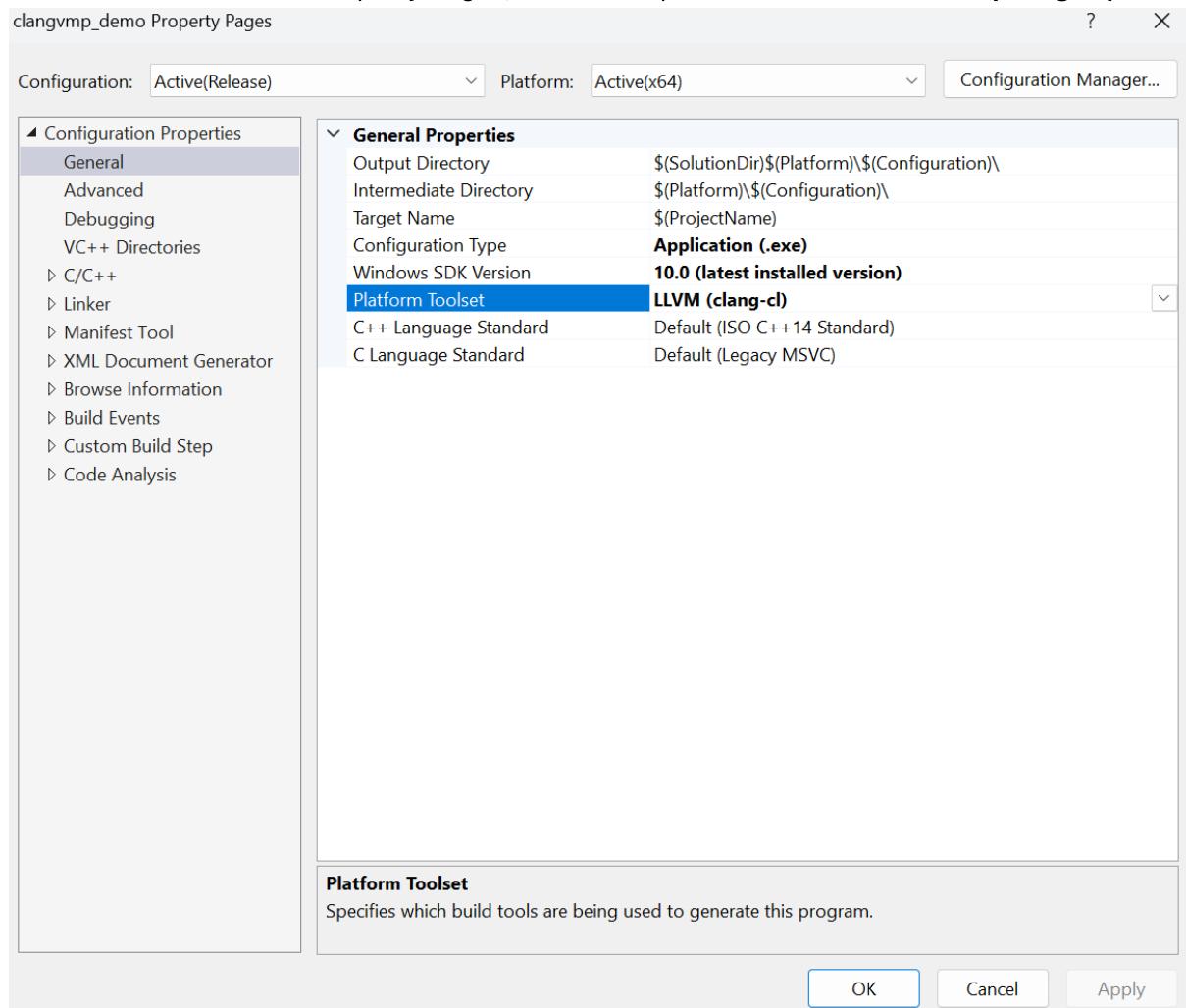
To apply this llvm like toolchain to Visual Studio project, you should do the following steps:

- 1. Copy the **Directory.build.props** to your **.sln** located directory, it's a xml file used to let Visual Studio locate this llvm toolchain:

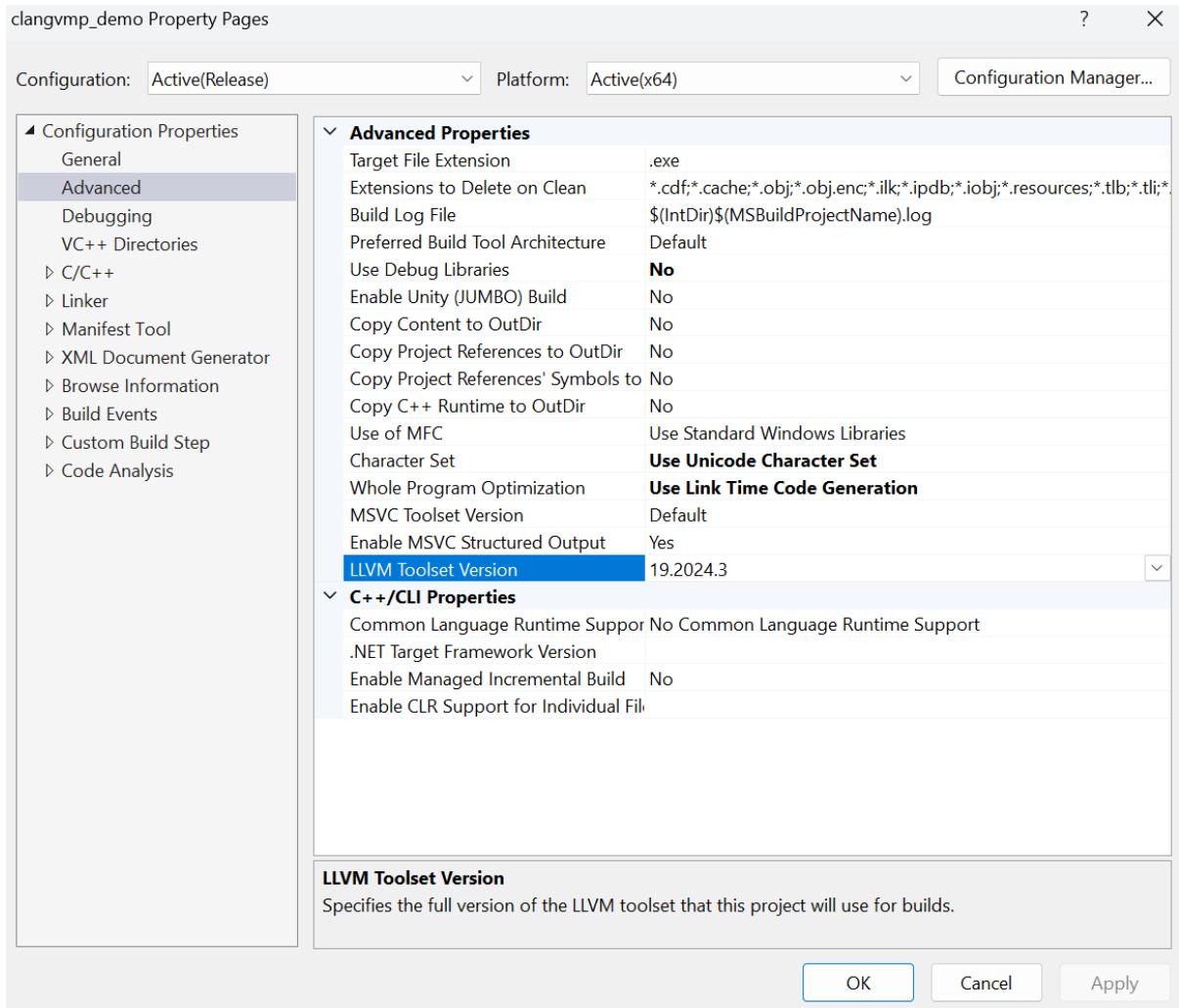
```
<Project>
<PropertyGroup>
    <LLVMInstallDir>/.../VPAssistant/product/clangvmp</LLVMInstallDir>
    <LLVMToolsVersion>19.2024.3</LLVMToolsVersion>
</PropertyGroup>
</Project>
```

Name	Date modified	Type	Size
.vs	3/24/2024 4:18 PM	File folder	
clangvmp_demo	3/27/2024 9:59 PM	File folder	
x64	3/29/2024 1:24 PM	File folder	
clangvmp_demo.sln	3/24/2024 4:18 PM	Visual Studio Solution	2 KB
Directory.build.props	3/25/2024 5:24 PM	Project Property File	1 KB

- 2. In Visual Studio Solution Property Pages, set General/Platform Toolset as **LLVM(clang-cl)**:



- 3. In Visual Studio Solution Property Pages, set Advanced/LLVM Toolset Version as LLVM-Main-Version.Year.Month(e.g.: **19.2024.3**, this is defined by ClangVMP llvm toolchain):



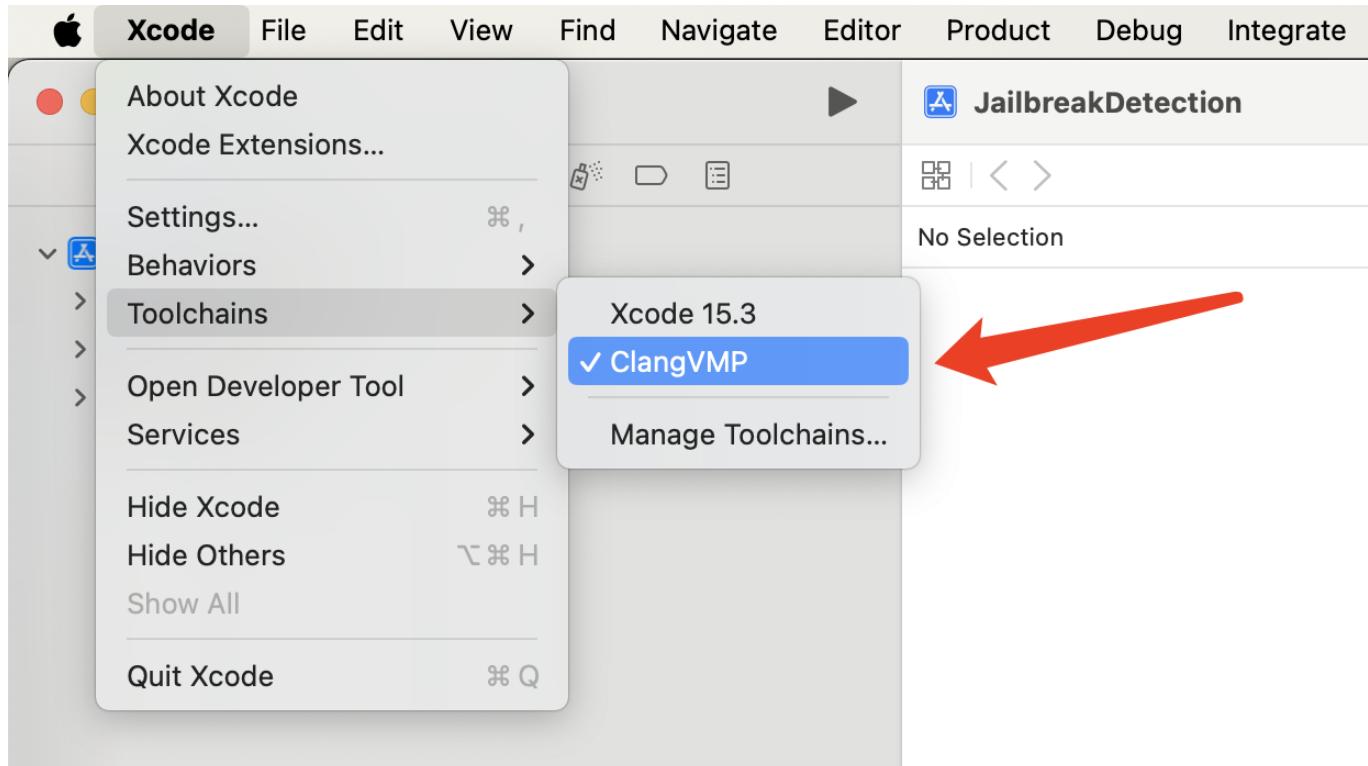
- 4. Make a test build to check whether everything goes well:

```

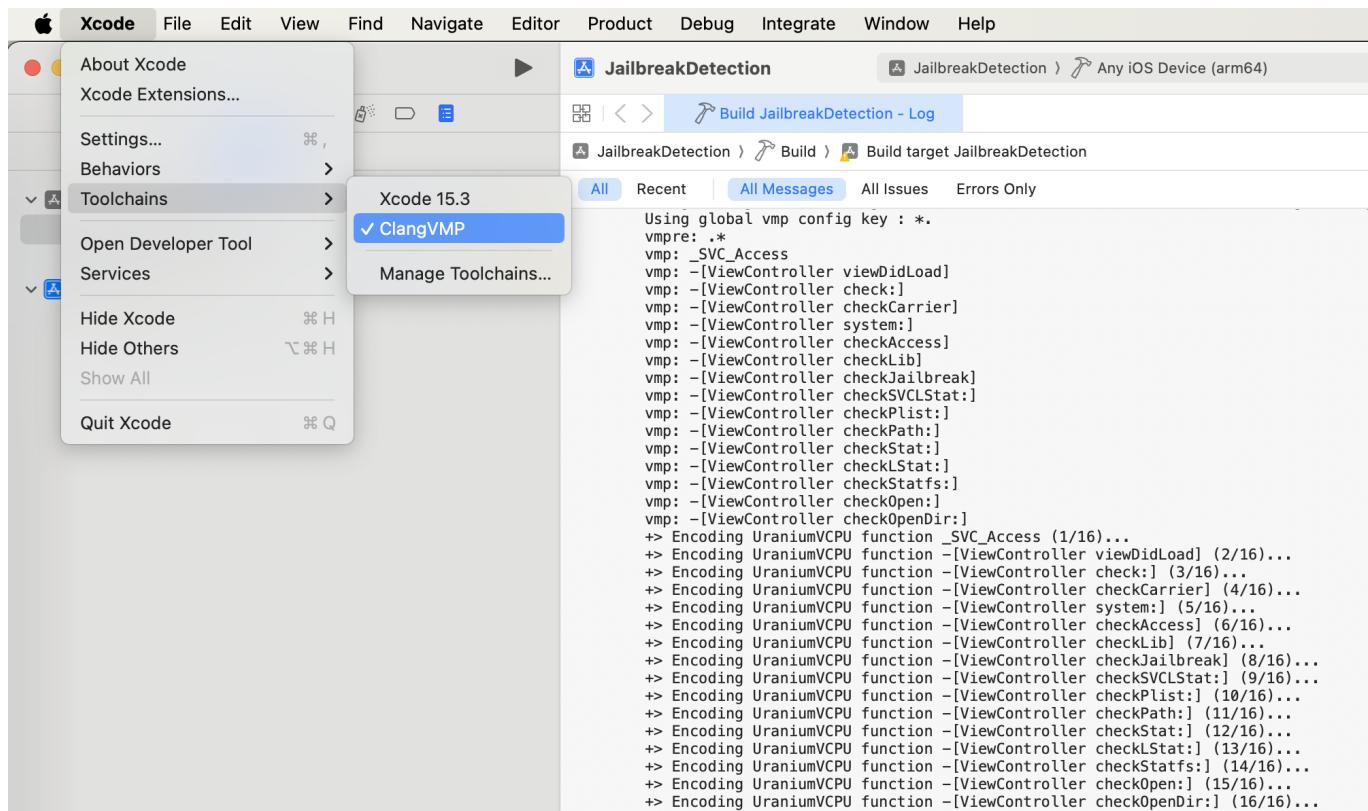
Output
Show output from: Build
Rebuild started at 5:54 PM...
1>----- Rebuild All started: Project: clangvmp_demo, Configuration: Release x64 -----
1>Using config file clangvmp.json.
1>Using source file vmp config key : clangvmp_demo.cpp.
1>vmpre: am_I_being_debugged
1>vmpre: do_main
1>vmp: ?do_main@@YAXHQEAPEBD@Z
1>vmp: ?am_I_being_debugged@@YA_NXZ
1>+> Encoding UraniumVCPU function ?do_main@@YAXHQEAPEBD@Z (1/2)...
1>+> Encoding UraniumVCPU function ?am_I_being_debugged@@YA_NXZ (2/2)...
1>clangvmp_demo.vcxproj -> C:\Users\geekneo\Desktop\Release\ClangVMP-Win64\demo\Visual Studio\x64\Release\clangvmp_demo.exe
===== Rebuild All: 1 succeeded, 0 failed, 0 skipped =====
===== Rebuild completed at 5:54 PM and took 03.924 seconds =====
|
```

Xcode

You can switch the Xcode and ClangVMP toolchain from main menu **Xcode/Toolchains** at anytime.



Make a test build to check whether everything goes well:



VMP Configuration

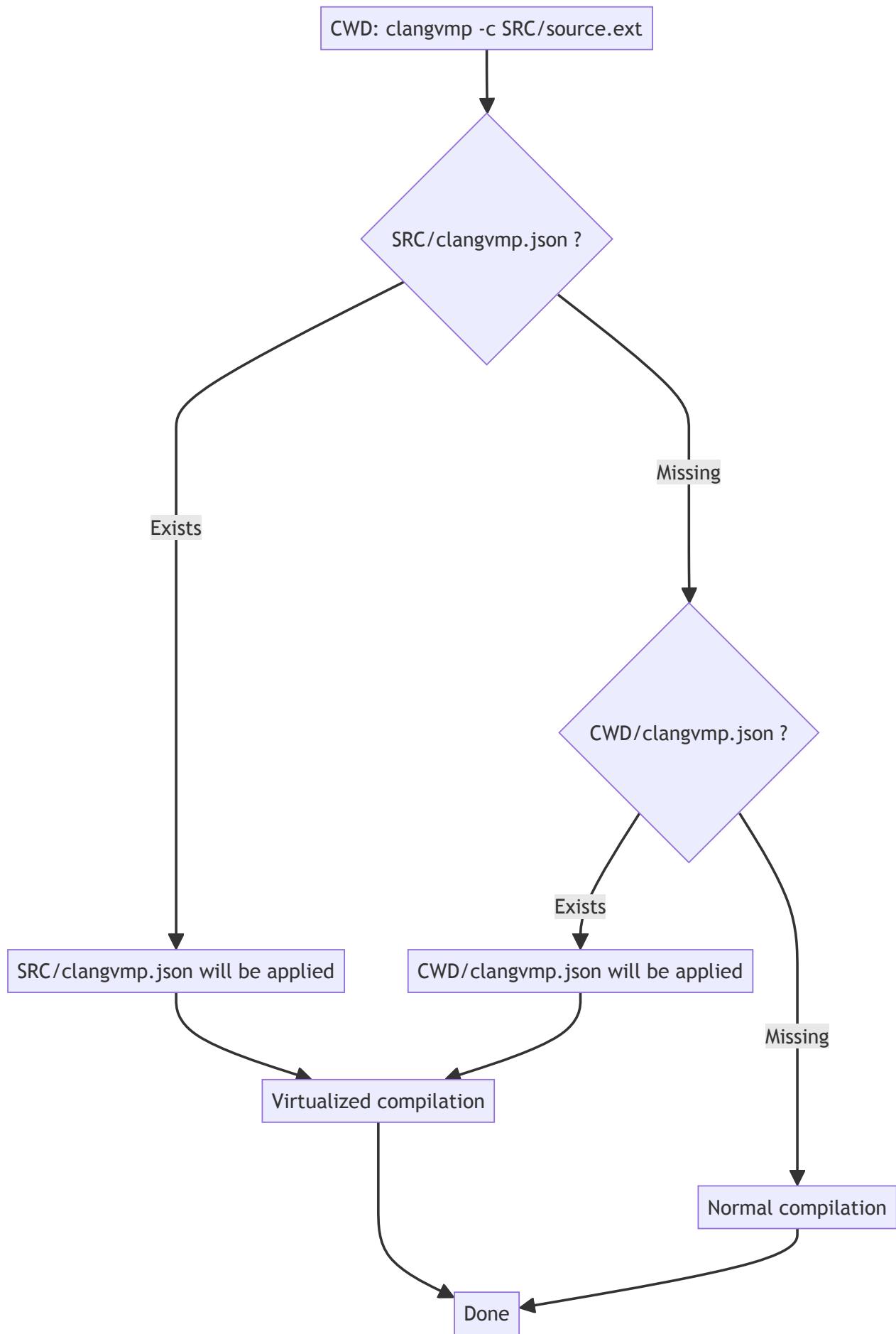
Unlike other Clang compiler-based source code encryption products such as **OL LLVM**, ClangVMP doesn't specify the encryption configuration using command line arguments or attribute annotations, but instead uses a **clangvmp.json** encryption configuration file to specify the encryption options. The generic format of **clangvmp.json** is as following:

```
{  
    "source_name.ext" : {  
        "vmpre" : ["symbol_regressor"],  
        "obfre" : ["symbol_regressor"]  
    }  
}
```

During compilation, the loading rule for the clangvmp.json configuration file is:

- **Firstly**, look for clangvmp.json in the folder where the input **source code file** is located, if found, use this configuration, otherwise go to **Secondly**;
- **Secondly**, look for clangvmp.json in the **current working path** of the compiler (for Xcode, this is the path where xcodeproj is located; for NDK, this is the parent directory of jni). If found, use this

configuration, otherwise turn to normal compilation;



source_name.ext: source code file name including its extension, such as helloworld.c, helloworld.cpp, helloworld.cc, helloworld.m and helloworld.mm. If this file name as a key in clangvmp.json, then the compiler will apply its symbol array to virtualize or obfuscate that matched function.

Key vmpre

vmpre(VMP regular expression): any function name symbol which matches this regular expression array will be **VMP virtualized**. e.g.: "*" means all symbols, "vmpfn" means symbols contain substring vmpfn. Visit [Regular Expression](#) for more information.

Key obfre

obfre(Obfuscation regular expression): any function name symbol which matches this regular expression array will be **OLLVM obfuscated**.

Demo

The following clangvmp.json means:

- All the function symbols in **UVMInterpreter.cpp** contain substring "interp" will be virtualized, contain substring "main" will be obfuscated.
- All the function symbols in **UVMRuntime.cpp** contain substring "exec" will be virtualized, contain substring "load" will be obfuscated.

```
{
    "UVMInterpreter.cpp" : {
        "vmpre" : ["interp"],
        "obfre" : ["main"]
    },
    "UVMRuntime.cpp" : {
        "vmpre" : ["exec"],
        "obfre" : ["load"]
    }
}
```

Raw vs ClangVMP

Here's the simple C/C++ code, let's see how the normal and virtualized compilation are going to generate the final assembly instructions.

```
static bool am_I_being_debugged(void) {
    FILE *fp = fopen("/proc/self/status", "r");
    char status[128];
    size_t rd = fread(status, 1, sizeof(status) - 1, fp);
    status[rd] = 0;
    fclose(fp);
    return strstr(status, "TracerPid:\t0") == nullptr;
}
```

```

int __attribute__((noinline)) do_main(int argc, const char **argv[]) {
    if (!getenv("NO_ANTIDBG")) {
        anti_debug();
    }

    // modify register to continue
    for (int i = 0; i < argc; i++) {
        printf(
            "You should change the register to quit this infinite loop %d.\n"
            "I'm being debugged(%s).\n",
            i,
            am_I_being_debugged() ? "true" : "false");
        sleep(2);
    }

    // what a nice day...
    puts("Have fun with A64Dbg UraniumVM virtualization debug mode~");
    return 0;
}

```

Raw

The normal compilation with clang compiler, we'll get the following assembly function:

```

do_main:0004 ; int do_main(int argc, const unsigned __int8 **argv)
do_main:0004                 EXPORT _Z7do_mainiPPKc
do_main:0004 _Z7do_mainiPPKc                      ; CODE XREF:
main+24↑p
do_main:0004
do_main:0004 ; __ unwind {
do_main:0004             SUB      SP, SP, #0xF0
do_main:0008             STP      X29, X30, [SP,#0x90+var_s0]
do_main:000C             STP      X28, X27, [SP,#0x90+var_s10]
do_main:0010             STP      X26, X25, [SP,#0x90+var_s20]
do_main:0014             STP      X24, X23, [SP,#0x90+var_s30]
do_main:0018             STP      X22, X21, [SP,#0x90+var_s40]
do_main:001C             STP      X20, X19, [SP,#0x90+var_s50]
do_main:0020             ADD      X29, SP, #0x90
do_main:0024             MRS      X8, #3, c13, c0, #2
do_main:0028 argc = X19                         ; int
do_main:0028             MOV      W19, W0
do_main:002C             STR      X8, [SP,#0x90+var_90]
do_main:0030             ADRP     X0, #byte_180@PAGE
do_main:0034             ADD      X0, X0, #byte_180@PAGEOFF ;
name
do_main:0038             LDR      X8, [X8,#0x28]
do_main:003C             STUR     X8, [X29,#var_8]
do_main:0040             BL       getenv
do_main:0044             CMP      W19, #1
do_main:0048             B.LT    loc_E8
do_main:004C             MOV      W20, WZR

```

do_main:0050	ADD	X27, SP, #0x90+ptr
do_main:0054	ADRP	X21, #byte_227@PAGE
do_main:0058	ADD	X21, X21, #byte_227@PAGEOFF
do_main:005C	ADRP	X22, #byte_239@PAGE
do_main:0060	ADD	X22, X22, #byte_239@PAGEOFF
do_main:0064	ADRP	X23, #byte_23B@PAGE
do_main:0068	ADD	X23, X23, #byte_23B@PAGEOFF
do_main:006C	ADRP	X28, #byte_1E7@PAGE
do_main:0070	ADD	X28, X28, #byte_1E7@PAGEOFF
do_main:0074	ADRP	X26, #byte_1E2@PAGE
do_main:0078	ADD	X26, X26, #byte_1E2@PAGEOFF
do_main:007C	ADRP	X24, #byte_18B@PAGE
do_main:0080 i = X20		; int
do_main:0080	ADD	X24, X24, #byte_18B@PAGEOFF
do_main:0084		
do_main:0084 loc_84		; CODE XREF:
do_main(int,char const**)+E0↓j		
do_main:0084	MOV	X0, X21 ; filename
do_main:0088	MOV	X1, X22 ; modes
do_main:008C	BL	fopen
do_main:0090	MOV	X25, X0
do_main:0094	ADD	X0, SP, #0x90+ptr ; ptr
do_main:0098	MOV	W1, #1 ; size
do_main:009C	MOV	W2, #0x7F ; n
do_main:00A0	MOV	X3, X25 ; stream
do_main:00A4	BL	fread
do_main:00A8	STRB	WZR, [X27,X0]
do_main:00AC	MOV	X0, X25 ; stream
do_main:00B0	BL	fclose
do_main:00B4	ADD	X0, SP, #0x90+ptr ; haystack
do_main:00B8	MOV	X1, X23 ; needle
do_main:00BC	BL	strstr
do_main:00C0	CMP	X0, #0
do_main:00C4	MOV	X0, X24 ; format
...		

If we decompile it in IDA or any other decompiler tool, we'll get a perfect pseudo C source code implementation like this, nearly the original source code:

```

__int64 __fastcall do_main(int argc, const unsigned __int8 **argv)
{
    int v2; // w19
    unsigned int v3; // w20
    FILE *v4; // x25
    const unsigned __int8 *v5; // x2
    __int64 result; // x0
    unsigned __int64 v7; // [xsp+0h] [xbp-90h]
    char ptr[128]; // [xsp+8h] [xbp-88h]
    __int64 v9; // [xsp+88h] [xbp-8h]

    v2 = argc;
}

```

```

v7 = _ReadStatusReg(ARM64_SYSREG(3, 3, 13, 0, 2));
v9 = *(_QWORD *) (v7 + 40);
getenv((const char *)byte_180);
if ( v2 >= 1 )
{
    v3 = 0;
    do
    {
        v4 = fopen((const char *)byte_227, (const char *)byte_239);
        ptr[fread(ptr, 1uLL, 0x7FuLL, v4)] = 0;
        fclose(v4);
        if ( strstr(ptr, (const char *)byte_23B) )
            v5 = byte_1E7;
        else
            v5 = byte_1E2;
        printf((const char *)byte_18B, v3, v5);
        sleep(2u);
        ++v3;
    }
    while ( v2 != v3 );
}
result = puts((const char *)byte_1ED);
if ( *(_QWORD *) (v7 + 40) == v9 )
    result = 0LL;
return result;
}

```

ClangVMP

The virtualized compilation with clangvmp compiler, we'll get the following assembly function:

```

do_main:0004 ; __int64 __fastcall do_main(int, const char **)
do_main:0004                                     EXPORT _Z7do_mainiPPKc
do_main:0004 _Z7do_mainiPPKc                     ; DATA XREF:
.data.rel.ro..L.vpand_com_ClangVMP_0:0398↓o
do_main:0004
do_main:0004 ; __unwind {
do_main:0004             SUB      SP, SP, #0x400
do_main:0008             STP      X0, X1, [SP,#0x400+var_400]
do_main:000C             STP      X2, X3, [SP,#0x400+var_3F0]
do_main:0010             STP      X4, X5, [SP,#0x400+var_3E0]
do_main:0014             STP      X6, X7, [SP,#0x400+var_3D0]
do_main:0018             STP      X8, X9, [SP,#0x400+var_3C0]
do_main:001C             STP      X10, X11, [SP,#0x400+var_3B0]
do_main:0020             STP      X12, X13, [SP,#0x400+var_3A0]
do_main:0024             STP      X14, X15, [SP,#0x400+var_390]
do_main:0028             STP      X16, X17, [SP,#0x400+var_380]
do_main:002C             STP      X18, X19, [SP,#0x400+var_370]
do_main:0030             STP      X20, X21, [SP,#0x400+var_360]
do_main:0034             STP      X22, X23, [SP,#0x400+var_350]
do_main:0038             STP      X24, X25, [SP,#0x400+var_340]

```

```

do_main:003C          STP           X26, X27, [SP,#0x400+var_330]
do_main:0040          STP           X28, X29, [SP,#0x400+var_320]
do_main:0044          STP           X30, X17, [SP,#0x400+var_310]
do_main:0048          BL
vpand_com_ClangVMP_9d4bf2e9b10ef523_1
do_main:004C          MOV           X3, X0
do_main:0050          BL
vpand_com_ClangVMP_9d4bf2e9b10ef523_0
do_main:0054          MOV           X2, #0
do_main:0058          B             vpand_com_ClangVMP_entry

```

If we decompile it in IDA or any other decompiler tool, we'll get a fixed pseudo C source code implementation like this, which will always keeps this kind of instruction sequence, saving register context and then entering ClangVMP virtual machine runtime library:

```

__int64 __usercall do_main@<X0>(__int64 a1@<X1>, __int64 a2@<X0>,
__int64 a3@<X2>, __int64 a4@<X3>, __int64 a5@<X4>, __int64 a6@<X5>,
__int64 a7@<X6>, __int64 a8@<X7>, __int64 a9@<X8>)
{
    __int64 v9; // x9
    __int64 v10; // x0

    vpand_com_ClangVMP_9d4bf2e9b10ef523_1(a2, a1, a3, a4, a5, a6, a7, a8,
a2, a1, a3, a4, a5, a6, a7, a8, a9, v9);
    v10 = vpand_com_ClangVMP_9d4bf2e9b10ef523_0();
    return vpand_com_ClangVMP_entry(v10);
}

```

Oops, we can see nothing about this function, all of the raw instructions are encoded as ClangVMP runtime data. **As so, reverse engineering will be extremely hard. This is the core value of ClangVMP.**

Trial version

The default installation without a valid license is a trial version for **code virtualized compilation test**, the result **cannot run**. If you want to apply it to your real products, please contact us to buy a license.

```

ClangVMP@vpand.com android % ndk-build -B
[arm64-v8a] Compile++ : antidebug <= antidebug.cpp
There's no license file ~/VPAssistant/license/clangvmp.license, please
contact us to buy a license.
-----WARNING-----
This trial version of ClangVMP is just used for compilation test, the
result cannot run!
-----
Using config file jni/../../clangvmp.json.
Using global vmp config key : *.
vmpre: main
vmp: _Z7do_mainiPPKc
vmp: main

```

```
+> Encoding UraniumVCPU function _Z7do_mainiPPKc (1/2)...
+> Encoding UraniumVCPU function main (2/2)...
[arm64-v8a] Executable      : antidebug
```

Licensed version

If you want to upgrade to a licensed version, here's the steps:

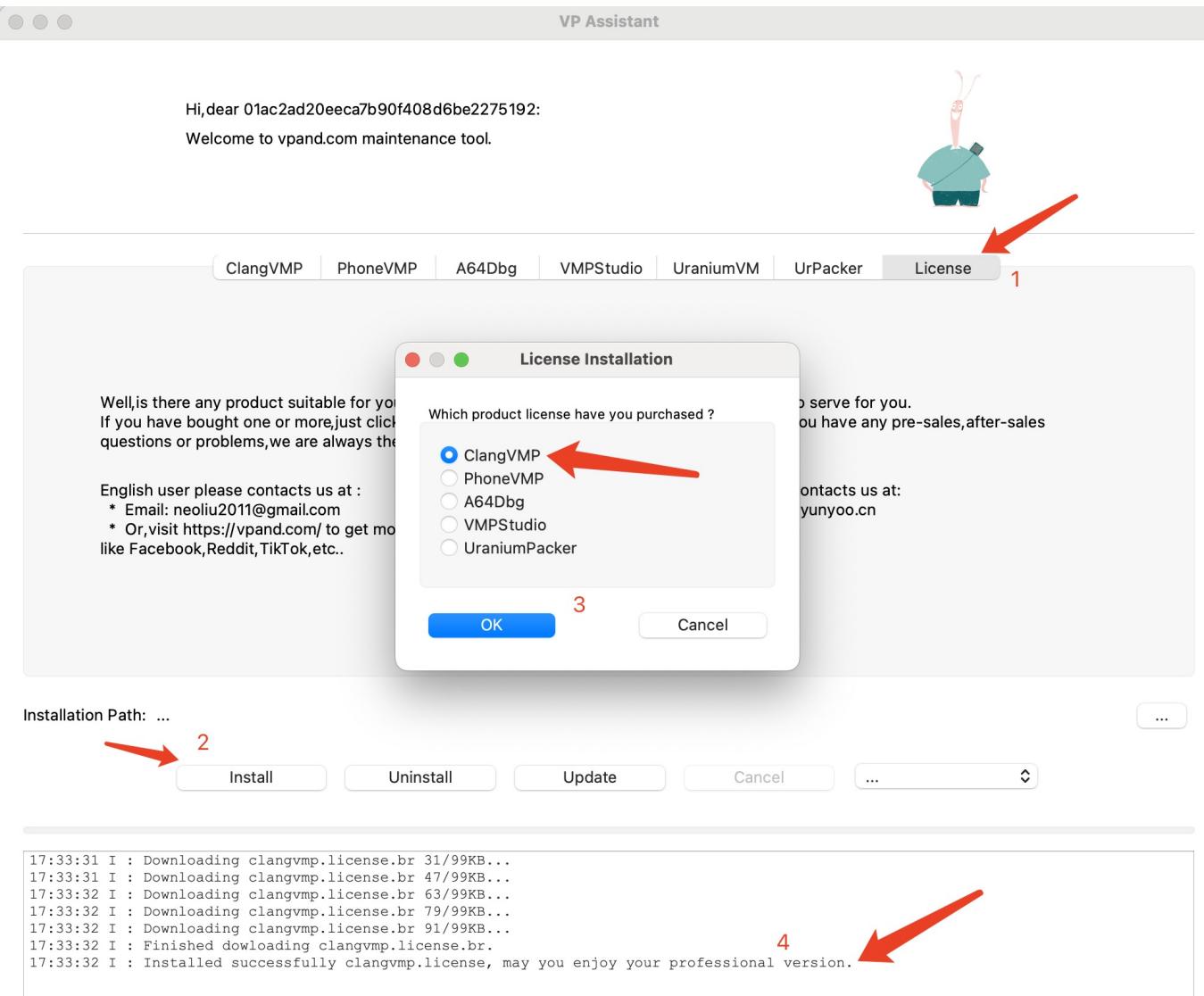
- 1. Contact us to pay for your license;

The price of ClangVMP: 9000\$/year/os-arch/license

- 2. Send your computer os-arch-hwid(copy from VPAssistant startup log) to us;

22:53:37 I : VP Assistant v1.1.2 (Apr 19 2024), current host hwid: mac-arm64-01ac2ad20eeeca7b90f408d6be2275192 .

- 3. Switch to VPAssistant License tab widget, click Install button to download and install the license file;



With this license, we can virtualized compile our final executable product:

```
ClangVMP@vpand.com android % ndk-build -B
[arm64-v8a] Compile++      : antidebug <= antidebug.cpp
Using config file jni/.../clangvmp.json.
Using global vmp config key : *.
vmpre: main
vmp: _Z7do_mainiPPKc
vmp: main
+> Encoding UraniumVCPU function _Z7do_mainiPPKc (1/2)...
+> Encoding UraniumVCPU function main (2/2)...
[arm64-v8a] Executable     : antidebug
```

Contact us

Email

If you have any questions or problems on our products or services, feel free to contact us via email at anytime:

- neoliu2011@gmail.com

We-Media

Till now, we-media is our main operation method, you can also contact us via the following platforms:

- [Facebook](#)
- [YouTube](#)
- [Reddit](#)
- [TikTok](#)
- [Instagram](#)

FAQ

Does ClangVMP support 32 bits architecture like x86 or arm?

In fact, we have finished the development of 32 bits architecture supporting. But besides some embedded system, mobile and desktop platform are nearly all x86_64 or arm64 architecture, so we haven't released it to current ClangVMP version. If you do need this kind of feature, please let us konw.

What's the price of ClangVMP license ? Could the license be purchased by quarter or month?

The price of ClangVMP is **9000\$/year/os-arch/license**, os means **Android/iOS/macOS/Linux/Windows**, arch means **x86_64/arm64**. You can visit [vpand.com](#) to check the latest price policy, all of our products are sold at expressly marked price, and the license term is by year. So till now, quarterly or monthly purchasing is not supported.

When the license is expired, can the code virtualized before run normally?

Yes, of course, the license is for ClangVMP compiler itself, it has no effect on the compiled code.

Can I update the license to rebind a new computer?

In principle, this is not supported. Because all of our license are offline type, it'll never connect to our server to check the license status. So if a license is released, it can not be repealed. This is good for client, as many clients are worried about data leakage. In order to avoid data leakage, offline license is the one and only choice, because we'll never read 1 byte data for our server to check what we need.