A New Architecture and…Names, Lots of Names

In the last decade or so, Apple has been slowly transitioning their entire product range to a new architecture that is, at its core, a rename of PowerPC to Apple Silicon. In this new architecture, the primary driver is the silicon. Apple have, of course, been using their own custom-made ARM chips in iOS for years, designated with an ‘A’ and a numerical specifier, such as ‘A14’. But Apple’s first generation of non-Intel devices are also known as “M1 Macs”.

The primary difference between the original Rosetta and Rosetta 2 is when translation takes place. In Rosetta, the translation happens ‘Ahead-of-Time’ (AOT), meaning that the software is translated before it is run. This is more efficient and faster than the translation that happens ‘Just-in-Time’ (JIT) after the software is run.

Rosetta is a translation layer that bridges the gap between the new Apple Silicon and the old Intel architecture. It allows software running on the new Apple Silicon to run on the older Intel architecture as if they were running on the same platform. This is known as translation.

Enter Rosetta 2 and the Universal 2 file format. With the M1 Macs, Apple took a similar approach: there’s a Universal 2 binary format that developers can use to work with both Apple Silicon and x86_64 architectures. However, Rosetta 2 doesn’t work in quite the same way as the original Rosetta, and this has both performance and security consequences.

Native M1 Software Is More Secure By Design

Although there are other barriers to clear for an attacker trying to carry out such an attack, it nevertheless remains the case that native code is inherently safer on an M1 Mac than translated Intel code, which can run without any code signing checks at all.

The benefits of running security software natively on Apple silicon are clear for both performance and security. You can easily verify this on an M1 Mac with a simple ‘hello world’ program. If we first compile the program below as arm64, you can see that the program runs much faster than when compiled as x86_64.

```
#include <stdio.h>

int main() {
    printf("Hello, World!
"%%n"n"n"
    return 0;
}
```

This fact is noted in Apple’s own documentation, which states, “You can maintain software applications that run natively on your M1 Mac. This enables you to maintain performance that is very close to the hardware, which is important for professional and creative workflows.”

However, Rosetta 2 doesn’t work in quite the same way as the original Rosetta, and this has both performance and security consequences. Any code that is not compiled as arm64 or x86_64 will run through Rosetta. This presents a potential attack surface.

Although all the components are locked, but there is still much more to be learned. For example, although Apple silicon Macs are designed to support the ARM architecture, some software modules require translation to run. This can be a performance issue, as translation is going to incur a performance penalty compared to a security solution that’s running native.

This allows for the possibility of software tampering: you can easily sign the code with a valid code signature, then compile it as arm64 or x86_64, and then remove the code signature. This program is then translated through Rosetta without the valid developer’s code signature. This allows for the possibility of software tampering: you can easily sign the code with a valid code signature, then compile it as arm64 or x86_64, and then remove the code signature. This program is then translated through Rosetta without the valid developer’s code signature.

In conclusion, while Rosetta 2 has some performance benefits, it also introduces security risks that need to be taken into account. The benefits of running security software natively on Apple silicon are clear for both performance and security. You can easily verify this on an M1 Mac with a simple ‘hello world’ program. If we first compile the program below as arm64, you can see that the program runs much faster than when compiled as x86_64.

```
#include <stdio.h>

int main() {
    printf("Hello, World!
"%%n"n"n"
    return 0;
}
```