When we interact with computers, we need some way of knowing that they are running real software. We would like to know this by minimizing our reliance on trust.

For example, when entering a password into a laptop, it would be nice to be sure that the laptop is running the software we expect it to be running. Unfortunately, there are many attacks that change the software running to something that we do not expect:

- An adversary may install bad software onto the laptop, such as a keylogger malware.
- A different app may be launched than the one that the user believes.
- An adversary may inject malware into a real app’s libraries, by tricking or coercing a developer.
- An adversary may trick the software update process, converting a real piece of software into a malicious one.

These attacks can happen anywhere along the path from a developer’s text editor to a user’s computer.

1. Developers write code
2. Applications include third-party libraries
3. Applications are built and packaged
4. Binaries are distributed over the network
5. Users download new software
6. Users download software updates
7. Users launch applications on their devices
8. Running applications interact with remote servers running code.

1 Library Imports

1.1 Example: Go Library Imports

In the Go programming language, libraries are imported by specifying an HTTPS URL from which the package should be downloaded.

    import "github.com/grpc/grpc-go"

When compiling the code, the developer PC will contact the server at the given URL over HTTPS (verifying the server certificate via TLS) and download the software bundle. On the other end, when a library
developer wants to update their library, they do so by interacting with the hosting server via HTTPS and whatever authentication the server has set up—credentials, maybe two-factor authentication, etc.

This has some good features: the server name is explicit so there is no ambiguity about packages and the decentralized nature of specifying individual URLs avoids the necessity for a central server that attracts attacks. However, this requires trusting the server hosting the library to secure the update process and distribute software honestly.

1.2 Example: Rust Library Imports

In Rust, library imports use a package manager called cargo. To add a library, a developer issues a command like the following:

```
cargo add rand
```

This adds another hop to the Go model: when compiling code, a developer PC will contact the central server crates.io to download the library. To update software, a developer will first update their code on some server such as github.com and then crates.io will contact the source server via HTTPS to update its own version. In some ways, this seems a bit more risky than the Go plan. There is now a central server managing all packages for the language, making an attractive target for attacks. The library developer is also now further away from the code that will use the library, adding an additional trusted party in between. However, this addition of a central repository has benefits as well: it provides a single place where all packages can be seen, and is a central server that makes auditing much simpler.

1.3 More Explicit Trust

In each of these, the end developer has to trust either a server like Github or a repository and a server like Github in addition to the actual library developer. If either of these servers becomes compromised, an adversary can modify the library that will be downloaded. To avoid some of this, one plan would be to have the library developer sign the software using their public key and include the signature with the downloaded software. To verify that the software was not modified by an intermediate party, the application developer’s PC can check that the signature is valid.

Of course, with any signature-based plan the mechanism for public key distribution is crucially important. In the software distribution case, the only reasonable plan is likely a Trust-on-First-Use based one which accepts the first public key it sees but verifies that future software updates use that same key. This protects against an adversary
taking control of, for example, the application’s Github repository after the end user installs the software once. However, key management is hard, so this is not widely used in practice.

2 Building Binaries

In order to run software on our computer, it is necessary to convert the source code (which is manually auditable) into a binary that is much more difficult to audit. Since compiling software is computation-heavy, software is typically compiled once and the binary is distributed to users. Without some way of verifying that a given binary came from a given codebase, this means that anyone who downloads this binary must fully trust the computer that generated the build.

2.1 Reproducible Builds

The only approach to this problem that has gained any ground is called Reproducible Builds. Typically, compilation is nondeterministic—the generated binary will often depend on randomness and traits of the computer that built the binary. However, there has been a recent push to redesign compilers to allow an independent party to recompile a given set of source code and verify that the generated binary is bit-for-bit identical to the widely published binary. This makes auditing much simpler.

3 Installing & Updating Software

Once a binary exists, the next step of the process is to distribute that software to user devices.

3.1 Application Developer Signs Package (Android Apps)

One possible option is to have application developers sign a package and attach a signature to the package. This way, it does not matter how the package is distributed to users: a user can download an application bundle from any server and know that it came from the developer who owns the corresponding secret key.

This is effective for verifying that future updates to the software came from the same developer. Note that it does not guarantee freshness: once signed, a package is always valid.
3.2 Repository Signs Packages

For systems with a central repository, another plan is for the repository to sign packages. This again allows the distribution itself to be untrusted—a repository could use a CDN to distribute packages without trusting that CDN to preserve the package contents.

Many Linux package managers, such as apt, pacman, and rpm, use a plan like this.

3.3 Third-Party Validator Signs Packages

Yet another option that does not require a single central repository is to have a trusted validator sign packages. This involves sending the source code and package to a third party, who will then perform some inspection of the package and, if it deems a package to be worthy, provide some signature over that package that verifies that the validator thinks the package is trustworthy.

This plan is used widely in practice. On Android, there is no requirement to install apps from the Google Play Store, but Google provides a service that inspects packages and attaches these signatures if the package passes. Similarly, Windows uses a validation plan for its device drivers.

3.4 Binary Transparency

One different plan to help involves an audit log that keeps track of all published binaries.

This helps prevent in particular targeted attacks—for example, if some adversary has a specific target in mind and compromised the distribution of the Linux kernel, they would likely be immediately noticed if they introduced a backdoor into Linux for the whole world. However, if they were able to introduce a backdoor and distribute that backdoored version only to their target, the adversary would be much more likely to evade detection. If clients check their received binary against the publicly available one before installing it, this personalized attacks can be avoided—if the attacker wants to change the binary for someone, they will need to change it for everyone.

4 Booting the System

In order to actually run an application, we rely on large amounts of software running on our computer, from the applications themselves to the operating system that supports them. If the operating system itself is compromised, for example, the modified OS could undermine all of the defenses we just discussed. One plan to help take
care of this involves a hardcoded root of trust baked into the CPU hardware itself and a hardcoded boot ROM that cannot be updated (hence ROM, for Read Only Memory). Booting then involves several steps:

1. On boot, the CPU will begin running the hardcoded Boot ROM code, which knows a hardcoded \( v_{kROM} \).

2. The Boot ROM will load the code for another layer called the *boot loader*. The boot ROM will then verify that the boot loader code was signed by \( v_{kROM} \). If the signature is valid, the boot ROM code will jump to begin executing the boot loader.

3. The boot loader will load the code for the operating system and verify that it was signed by \( v_{kbootloader} \). If the signature checks out, the bootloader will redirect control to the operating system.

This way, the system can verify that only boot loaders that the hardware manufacturer trusts are executed, and those boot loaders can verify that only trusted operating systems are executed.

This plan is used in many systems, from iPhone and Android to chromebooks and UEFI secure boot on PCs.