Agnostic embedding of digital signatures into programs

May, 2020

Sean Liao
M.Sc. Security and Network Engineering
University of Amsterdam
Amsterdam, The Netherlands
sean.liao@os3.nl

Davide Pucci
M.Sc. Security and Network Engineering
University of Amsterdam
Amsterdam, The Netherlands
davide.pucci@os3.nl

***Abstract*—Mandatory Access Control (MAC) system is the strategy on which the Operating System (OS) relies to grant, restrict or deny access to specific resources. In this document we evaluate the impact of a MAC that discriminates processes, based on the signature the originating binaries and/or scripts are shipping with: the signing approach proposed is the so-called suffix-based, *i.e.*, the signature is appended as a tail to the program file itself, rather than relying on further system dependent storage areas. The strategy is enforced using the fanotify Linux kernel Application Program Interface (API), in order to intercept the originating processes and make them all flow through a single point, so to verify them before letting them run normally. We show how the suffix-based approach, unlike the traditional approaches relying on File System (FS) capabilities, is resilient to the cross-system transmission of the programs files, although still in the need of a more robust backing system to enforce it.**

***Index Terms*—digital signature, signing, fanotify, eBPF, MAC**

I. Introduction

Back in the early days of Information and Communications Technology (ICT), it was not much of an issue to not care about programs trust. Due to very limited ways of accessing devices, when what we nowadays call Internet was not prominent yet and the widely used way of interacting with them was to physically sit next to them, security constraints did not represent that much of a topic as it is today, instead. The Operating System (OS) was not thought to base its usage with dealing with foreign entities, or be modularly extended with new programs, drivers and applications.

During the last thirty years, though, a lot has changed and the usage that was being initially done with such devices does not make that much of a sense anymore: if a device is not connected to the Internet, it is not worth to be used at all. Connectivity has become a major — if not the only big one — requisite, as it represents the mean through which all the modern — technology-related and not — activities are done.

This break change brought along a new side requisite: the ability, from the OS perspective, to discern what is trustworthy and what is not. In fact, relying on connectivity, devices can be extended and customized, installing and removing applications and customizing general OS behavior. Although this represents a very much appreciated development, it brings the drawback to be possibly exploited to for data exfiltration, damaging the system, and so on. Back to the point, the device manufacturer and more specifically the OS has to be able to prove the trustworthiness of every program intention, today more than ever.

The most traditional approach provides trust relying on a Public Key Infrastructure (PKI): any program is verified using public and private key pairs. The authors of the program or the OS manufacturer itself use their private key to sign it, and the OS takes care of verifying the signature using their public key. This verification is enforced building a Mandatory Access Control (MAC) system: such a system, in fact, on behalf of the OS, constrains processes, by allowing or disallowing certain functions or actions range, by granularly discerning which operations programs are allowed to do, or just by discriminating which programs can be run and which not at all.

II. Problem

Traditional MAC approaches are heavily dependent to the location in which the signature data is placed. Many put this information directly in the binary, supposing that all processes are only generated by binaries, using areas in the binary layout format that the OS has predisposed for. Yet processes are not always generated by binary files. Nowadays, the the conception of program is way broader than before: a program can also be a simple script, which is relying on an interpreter — which is a binary, instead — to be run. In this case, which is the signature to be verified? The interpreter can be trustworthy, but it does not mean the script which it is going to run is, too. Then, how to trust the program? And how to sign it?

Also, with virtualization techniques constantly growing by usage, how does this topic applies to containerized applications [1] or even newer strategies such as unikernels [2]? More and more techniques and execution approaches are getting used and still a proper wider approach which can hopefully apply to all those areas is to be identified and/or standardized.

In fact, depending on the area in which the signature is to be inserted, different ways are there to tamper the program and extract the signature, eventually stripping it off, exploiting too permissive OS policies. Furthermore, manufacturers which are proposing and distributing a different OS are — in most cases, not fully — adopting different approaches, using different areas in which to put the signature information, *e.g.*, Linux distributions relying on files Extended Attributes (xattrs), functionality offered specifically by the File System (FS).

This research tries to deepen this lack proposing a novel approach to the problem, which consists of appending data onto the end of the program file, without further needs of dedicated areas in the binary format specification. The same approach is applicable for scripts as well, consisting of payloads appended under the shape of a comment, at the end of the script file itself. The signature this way, regardless of the shape of the program, is always available, sticking to the execution unit that is going to be run, eventually guaranteeing cross trustworthiness between different and co-interacting program files. Furthermore, such an approach would survive to most — if not all — transport methods, *e.g.*, downloading over HyperText Transfer Protocol (HTTP).

This solution resemble what the DigSig [3] and bsign [4] projects have been trying to do, extending it to a more varied environment, which involves different programs shapes.

III. Related Work

Relevant prior work in this area consists of different approaches to whitelisting applications based on various available attributes.

On Windows systems, AppLocker (available starting from Windows 7) has the option to limit execution based on application path, execution context and embedded signatures. Additionally, there are various third party applications which provide similar functionality:

- Ivanti Application Control
- McAfee Application Control
- Trend Micro Application Control
- Faronics Anti-Executable
- Kaspersky Whitelisting
- Airlock Application Whitelisting
- Thycotic Application Control

On macOS, System Integrity Protection [5] limits access to various parts of the filesystem to Apple signed executables, while Gatekeeper [6] enforces application signatures since macOS 10.7.3. There are recent reports [7] that macOS 10.15 has increased the number of checks, incurring network access for notarization checks for shell scripts as well.

Linux provides several different frameworks for adding security checks. The fanotify [8] API was introduced sometime around 2009 with the goal of providing a stable, supported method of integrating malware scanners [9]. This has been used by RedHat for their File Access Policy Daemon [10] for path based whitelisting and integration with SELinux.

Linux Security Module (LSM) provides a modular system for creating kernel modules with more customizable checks. AppArmor and SELinux are perhaps the two best known ones, developed by Canonical and RedHat respectively. AppArmor is primarily path based, while SELinux policies relies on information store in filesystem xattrs. Microsoft recently (2020) announced their Integrity Policy Enforcement [11] LSM, for hash based integrity measurement (this may overlap with IMA described below). Between 2002 and 2009 there was DigSig [3], which enforced application signatures based on bsign [4]. bsign stored the signatures in an extra Executable and Linkable Format (ELF) section.

The kernel Integrity Measurement Architecture (IMA) [12] project has started in 2005 to provide a trusted chain of integrity checks and remote attestation [1] with Trusted Platform Module (TPM). Metadata for the checks is stored with xattrs. Early mailing list discussions mentioned the possibility of modifying interpreters to enforce IMA checks [2].

At a higher level, for Docker containers, there is Docker Content Trust (DCT) [13] for containers signing and policy enforcement during build and execution. Binary Authorization [14] is the service for Google Cloud Platform (GCP) available since 2018, which integrates signature checks at various points within their cloud compute services.

IV. Research Question

This section presents our main research question and sub-questions that follow from it.

> Does the suffix-based signature approach represent a feasible and effective solution for storing and validating signed code?

This question can be divided into these sub-questions:

1) How to enforce the policy for compiled executables?
2) How the enforcement differ in case of interpreted runtimes?
3) How does the suffix-based approach compare to already existing solutions?
4) Can the policy enforcement be plugged as an extension to existing frameworks?

V. Experiment Design & Implementation

The hypothesis to be tested is that signatures can be attached directly to arbitrary executables and scripts in a way that 1) can be used for verification 2) is not easily lost 3) and does not affect normal operations of said executable/script. Due to the wildly different Application Program Interface (API) needed to test this on the various operating systems, the proof of concept is limited to Linux based systems.

A. Signature algorithm

The first step is to select a signature algorithm and tooling. Several options have been considered here: Rivest-Shamir-Adleman (RSA) with GNU Privacy Guard (GPG); either RSA or Elliptic Curve Digital Signature Algorithm (ECDSA) with OpenSSL; either RSA or ECDSA or Ed25519 with OpenSSH;

[1]https://keylime.dev/

[2]https://marc.info/?l=linux-kernel&m=111682497821375&w=2

Ed25519 with Minisign and Signify. Given the standing recommendation not to use Pretty Good Privacy (PGP)/GPG [15], this left three options only. OpenSSL's tooling counts as straightforward, but shares the swiss army knife design of GPG. OpenSSH signing and verification is more opinionated and less flexible, but allows for the use of Secure Shell (SSH) keys and comes with its own authorized signers system. Minisign and signify are two compatible tools developed for OpenBSD [16]. They only support Ed25519 keys and are thin wrappers around the algorithmic signing primitives. This is in line with recommendations to not use RSA [17] and ECDSA [18].

The signature has hence been based on the signify, due to both its simplicity and the small signature size which made embedding slightly easier. The proof of concept [3] uses a pure Go (language of choice) library implementation of signify [4] which wraps the signing function in the standard library [5].

B. Signature embedding

The goal is to have a signature that stays with the executable/script. This rules out a separate signature file which can easily get left behind or lost, or storing signatures in the xattrs of the FS which are not preserved over most transport protocols and not available on some FSs.

Almost all binary executables are ELF formatted, which is highly structured. bsign modified executables by adding an extra ELF section to store its signature, but the extra section has been observed to be unnecessary and arbitrary data can be appended to the end and it will get ignored. For scripts, relying on the escape sequences to denote comments offered by most scripting languages, the signature can be added as a comment line at the very end of the script.

The final signature has the following format:

```
\nPrefix:MagicString:OriginalLen:Base64Signature:\n
```

1) `Prefix` is an optional string for comment characters, ex. `"# "` for bash or python scripts.
2) `MagicString` is currently the literal `AUTHSIGv0` used as aide in locating the signature line.
3) `OriginalLen` is the length of the original input file in bytes as an ASCII decimal number. This is necessary since the exact input is needed to the verifying function.
4) `Base64Signature` is the base64 encoded ed25519 signature, same as the output of signify.

C. Intercepting and enforcing

Given a system conscious of who only is to be trusted, the trust has to be practically applied against what a consumer is asking the system to run. In order to do that, the strategy introduced above has to rely on a middleware to do the effective enforcement: this is a layer, *i.e.* a daemon, which intercepts every action in the user space which involves an execution. This entity sits between every user-requested execution and the OS which is going to effectively give life to that request. Clearly, the term *execution* is very vague and has to be somehow parameterized within a collection of clear and specific measures that the OS can understand. Mainly, when speaking of executing, what is being referred is the `execve` syscall: calling such a procedure, normally, the OS will run arbitrary code in the context of an already existing process — *i.e.*, the caller —, whilst the memory allocation is completely replaced in favor of the requested program. From now onwards, the *exec syscall* term will be used to indicate what has been explained above. To enforce a policy against programs, the middleware has to sit between the exec syscall request and its effective resolution, making choices whether it can be done or not, prior to the kernel itself. If and only if the target of such invocation is part of the trust-zone, then it is allowed to proceed; in any other case, the execution must be denied.

Such an architecture can be achieved relying on different frameworks or entry-points mentioned in the Related Work section. Given the nature of the project and the lack of time, a constraint has been applied for this choice: ignoring all the frameworks which involved extending the kernel. This massive scope reduction led to consider only two options: make the interceptor rely either on fanotify or extended Berkeley Packet Filter (eBPF).

The fanotify API can be used to monitor all the objects in a mounted file system, enabling a fine-grained access permission decisions, if needed, analyzing the content of the objects themselves. More specifically, to make it adapt to the proposed use-case, initializing a fanotify group with few specific flags is needed: 1) `FAN_CLASS_CONTENT`, which is intended for event listeners that need to access files when they already contain their final content 2) `FAN_NONBLOCK`, so that reading from the intercepted file descriptor will not block.:

```
mask, err := fanotify.Initialize(
    unix.FAN_CLASS_CONTENT|unix.FAN_NONBLOCK,
    unix.O_RDONLY|unix.O_CLOEXEC|unix.O_NOATIME)
```

On the other hand, the defined group has to mark as to be tracing all the events regarding files opened for executions in the scope of a given FS (the FS mounted on the system root, *i.e.*, `"/"`, in the example below):

```
err := mask.Mark(
    unix.FAN_MARK_ADD|unix.FAN_MARK_MOUNT,
    unix.FAN_OPEN_EXEC|unix.FAN_OPEN_EXEC_PERM,
    unix.AT_FDCWD, "/")
```

More specifically, 1) the `FAN_OPEN_EXEC` flag allows to trace when a file is opened with the intent to be executed, while 2) the `FAN_OPEN_EXEC_PERM` one when a permission to open a file for execution is requested.

With such mask, fanotify can intercept all the aforementioned events, passing several information linked to the event traced:

```
struct fanotify_event_metadata {
```

[3]signing/verifying code: https://github.com/seankhliao/uva-ot/tree/master/sig

[4]https://github.com/seankhliao/signify

[5]https://golang.org/pkg/crypto/ed25519/#Sign

```
    __u32 event_len;
    __u8 vers;
    __u8 reserved;
    __u16 metadata_len;
    __aligned_u64 mask;
    __s32 fd;
    __s32 pid;
};
```

Relying on 1) the `mask` field, it is possible to check what mask flag generated the specific event 2) the `fd` field, it is possible to query the file descriptor wrapping the file whose access has been requested 3) the `pid` field, it is possible to access the specific process which issued the exec syscall.

As such, the interceptor is able to query the trust-zone to check whether the file associated to the `fd` does contain a valid signature and process a response action accordingly. In fact, once the access event has been verified, a response resembling the following object is shipped to fanotify:

```
struct fanotify_response {
    __s32 fd;
    __u32 response;
};
```

Where `response` corresponds to one of the two constants exposed by the kernel `FAN_ALLOW` and `FAN_DENY`. Such proof of concept of the daemon [6] relying on fanotify has been written relying on fanotify's Golang bindings.

VI. Results & Discussion

A. Technical details

The executable or script is passed directly to the ed25519 signing function, without prehashing. Based on RFC 8032 [19] Section 8.5, it is not necessary and recommended against, given that it increases the surface area to include additional hash collisions. Section 8.7 warns against signing "large amounts of data at once", but most executables and scripts should be fine. Exceptions may include early implementations of `true` which was just and empty file, and large self-decrypting archives that do not fit in memory.

In most cases the signature line should be between 114 and 127 characters long, depending on the size of the file and the comment prefix. While it would have been nice to keep the line length under 80 characters, the added complexity was not considered justified. Additionally, it is not checked that nothing else exists after the signature line, though this may cause issues if a script tries to read itself from the end.

B. Signed executables

Verifying the signature and checking a hash/checksum both verify that the integrity of a file has not changed since the time the baseline measurement was taken. Where they differ is that a public key can be used to verify multiple files, whereas a checksum is needed for every file you verify. Ideally, the public key or hash checksum is transported in a secure channel out of band from the file you wish to verify, otherwise a compromise of the transport could also modify the verifier to mask any changes as pointed out in response to a Linux Mint website compromise [20].

[6]https://github.com/streambinder/oath

Additionally, code signing only places a check at the beginning of execution, it places no further restrictions on behaviour down the road. This means it is only effective in protecting the supply chain from the point of signing, either script creation or binary builds, to the point of execution, with fewer gaps in coverage compared to transport level encryption such as Transport Layer Security (TLS) or distribution package signing which ends once files are unpacked to the disk.

C. Public key selection

In verifying a signature, the public key is needed to check the signature against. There are multiple options, namely to check against all known public keys, to check against public keys authorized for files with a specific name/path (prefix), or to check against public keys authorized for a specific file using its hash as an identifier.

Checking against file identity introduces complications, as you now need to securely retrieve the checksum - public key pairings and update your local trust database. Furthermore, if you can already securely retrieve a trusted checksum then there is little added benefit to a signature check.

Indiscriminately checking against all keys is similar to the certificate checks for most websites under TLS, as in a compromise in any key could impact the trust of the entire system.

Path based key selection offers a middle ground on a stable attribute that is unlikely to change (for example the file name or the entire `/usr/bin/` directory) while still retaining some control over signature namespacing. It also has the added benefit of being more intuitive for users.

D. Interpreted runtimes

If the approach described till now is easily applicable in systems for which program is basically a synonym for binary, when it comes to interpreted runtimes and scripts, instead, the situation grows in a very complicated way. In fact, opening up to a MAC system able to regulate flawlessly and transparently both traditional binaries and interpreted programs brings in the need to cover much more use-case scenarios: the first big complexity point comes with the fact that whilst a binary, when being run, is always the element against which the exec syscall is run, for interpreted runtimes, also the interpreter binary could referenced in the syscall and the script itself is only read and run afterward by the interpreter itself. More specifically, the cases are enumerated below.

The first scenario is when the program is run normally by path:

```
$ path/to/script.sh
```

In this case, the interpreter is resolved using the *shebang* heading, and the same logic proposed for binaries applies, as the exec syscall is done, by matter of fact, directly against script path.

Second scenario is when the program is passed as argument to the interpreter:

```
$ bash path/to/script.sh
```

In this other case, the exec syscall is simply performed against the interpreter binary, in which case the signature verification is just a little more than completely useless. In fact, interpreters are means through which scripts instructions can be run: in this case, the operations to be constrained are decoupled from the binary, decoupled from the effective exec syscall. Hence, the information obtained by fanotify is pointless, as in the need of the process `argv` pointer, too, to analyze what exactly the interpreter is going to run and, then, to check the signature on the script itself. On this regard, fanotify events interception sit on a time lapse during which the file path requested to be run has not been read yet, not only the syscall has not been performed. This means that all the mappings the kernel is going to do to expose informations about the process — *e.g.*, data in `/proc` special directory — are not ready yet, hence not accessible from the policy daemon. In a common scenario such as when a script is run using a shell, usually the shell process is spawning a new process with the `fork()` syscall: the resulting child process will then perform the exec syscall to run the requested script; when the `FAN_OPEN_EXEC` event is collected, hence, the metadata mapping in `/proc/self` represents only information about the process which performed the syscall, *i.e.*, the process result of the `fork()` syscall, the forked shell. This is why traditional approaches to gather information about the process `argv` are not working in this case. Leveraging on the fact that, at some time, the interpreter process is going to perform a read against the script file passed as argument, a possible solution to circumvent the issue is to let the daemon start tracing also `FAN_OPEN_READ` events, *i.e.*, all the files being opened for a generic read. In fact, intersecting the information gathered using both `FAN_OPEN_EXEC` and `FAN_OPEN_READ` to first pinpoint a interpreted script execution scenario and then identify which script will be run, would enable the possibility to still verify the signature of the to-read script, eventually allowing or denying the read operation, instead of the exec syscall itself. What remains complicated, though, is how to identify a binary as an interpreter, without simply matching the name or parameters that could be simply tampered.

Third scenario is when the program is read from STDIN:

```
$ curl https://url/to/script.sh | bash
```

In this case, also tracing `FAN_OPEN_READ` events has no impact. In fact, there is no read operation whatsoever regarding the FS, other than the one to read the interpreter binary to be run. On the other hand, the interception mechanism gives access to process related file descriptors and using the `dup()` [7] syscall to duplicate the file descriptor `0` — *i.e.*, standard input — to overcome its one-consume-only nature, would enable to the possibility of analyzing the script even when a pipe to the interpreter is used to run a program.

[7]https://www.man7.org/linux/man-pages/man2/dup.2.html

Fourth and last scenario is when the program is passed inline as a flag:

```
$ bash -c 'echo from flag'
```

This very corner case offers no feasible solution, leading to block any interpreter to be run if not in one of the scenarios mentioned earlier.

E. eBPF counterpart

As an alternative to fanotify, an implementation of the daemon based on eBPF has been attempted. The useful consideration about eBPF is that it allows programs to be run no more (at least entirely) in the user space, but in the kernel's, on one hand reducing the overhead produced in collecting traced event and, on the other, greatly increasing the number of event types which are traceable.

Consisting BPF programs of two subprograms — the kernel space and the user space ones —, the first one collects, traces and exposes events to the latter, which can handle the information for further usage. To trace the the exec syscalls, the following code [8] has been used to be run in the kernel space, leveraging on the BPF Compiler Collection (BCC) toolkit:

```c
int syscall__execve(
    struct pt_regs *ctx,
    const char __user *filename,
    const char __user *const __user *__argv,
    const char __user *const __user *__envp)
{
   struct data_t data = {};
   struct task_struct *task;
   data.pid = bpf_get_current_pid_tgid() >> 32;
   task = (struct task_struct *)
       bpf_get_current_task();
   bpf_get_current_comm(
       &data.comm,
        sizeof(data.comm));
   data.type = EVENT_ARG;
   __submit_arg(ctx, (void *)filename, &data);

   int i = 1;
#pragma unroll
   while (i < 128 &&
       submit_arg(ctx,
           (void *)&__argv[i++], &data) != 0);
   return 0;
}
```

In the user space, on the other hand, using the BCC bindings for Golang, a new module is being created and the correct kernel syscalls are mapped to the functions exposed on the kernel space program [9]:

```go
module := bcc.NewModule(kProgram, []string{})
defer module.Close()
```

[8]Given that, in this specific case, the eBPF-based implementation represented the failed attempt to overcome the limits offered by the fanotify-based one, the whole code base will not be shown, but only the crucial parts. Nonetheless, the code is available at https://github.com/streambinder/oath

[9]The `kProgram` variable represents a string comprising the entire kernel space program for which the C code snippet represent the crucial portion

```

syscall := bcc.GetSyscallFnName("execve")
sysKprobe, err := module.LoadKprobe(
    "syscall__execve")
if err != nil {
    handle(err)
}

if err := module.AttachKprobe(
        syscall, sysKprobe, -1); err != nil {
    handle(err)
}
```

The impact on the performance of the switch to eBPF is noticeable, but on the other hand, eBPF only offers a way to trace events and not the straightforward native possibility to actively intercept, interact with and control them. As an alternative, what has been tested was an approach based on the combination of both a eBPF tracer with a signal-based process control mechanism. Basically, given an exec syscall traced by eBPF, several information can be gathered about the process and, relying on them, the dameon:

1) sends a `SIGSTOP` signal to break the process execution
2) performs the verification of the signature
3) eventually sends a `SIGCONT` or a `SIGKILL` depending on the result of the verification phase

Such an approach led to a solution not completely transparent to the user but, most of all, to a serious flaw: being the exec syscall performed straight away, it is not predictable which program instructions are executed in the time lapse needed for the daemon to send the no-more-preventive `SIGSTOP`.

F. Shortcomings of a non-LSM daemon

The implementation proposed relies on a daemon to enforce the policy: the interceptor, as already mentioned, is just a simple process owned by the root user and, therefore, given the proper rights, it can be easily killed, neutralizing the shield offered by the project. As a prerequisite, hence, the OS user privilege separation must be trusted.

Furthermore, the given approach is proven to be only able to reliably check the distribution part of programs: limiting programs actions range and being able to apply a more granular programs constraining requires more information only available replacing the current interceptor with a dedicated LSM. As mentioned earlier, this scenario has not been considered given the time constraints of the project.

G. Systems scenarios

The approach proposed is evidently leaving aside a substantial matter: the scenarios in which such a solution would be used. As a matter of fact, what has been evaluated is the bounds of the approach in terms of types of programs it can be stretched to cover and how to reach the same outcome and protection in the various different programs case, regardless of what is the intended use of the underlying system. But this obviously represents an important node in the discussion: the aforementioned approach used to protect systems intended for normal desktop usage will have different requirements than the ones imposed in a server context. As a matter of fact, a desktop system is more likely to be changing with time going, suffering several installation, removals and updates of software, and especially more likely to be serving third party software which could have never predicted at the time of the installation of the original system. On the other hand, in a server context, the system is by definition built to serve specific purposes only, rarely opening up for scope relocation by changing directly the live system, removing and adding new software. The only change impacting the software of a server is usually the update, and it is hopefully done only via a certified source. As described, the two use-case scenarios have very different needs: in the latter case, requisites can be relaxed, using the solution to block all unsigned binaries and scripts, and all the interpreted programs which can not be flawlessly evaluated, *i.e.*, all the scripts executions not based on the *shebang*, dropping a big slice of complexity from the logic.

H. Comparison

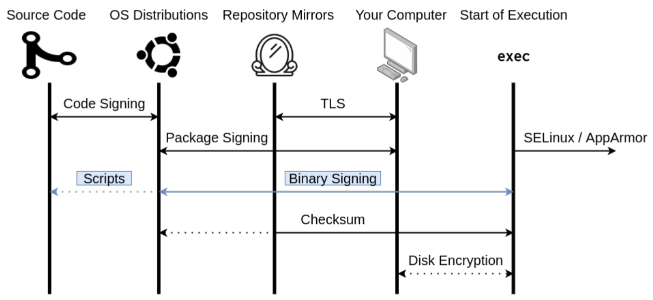

Fig. 1. binary distribution

Fig. 1 shows how source code is protected in the different stages from its upstream release to the moment of execution and beyond. While there is some overlap between the different security mechanisms, they may also not be directly comparable as they protect against different failure modes. Nonetheless, signing the final executable results in the longest unbroken chain.

Comparing to other software, the proposed solution is similar to RedHat's fapolicyd in that both limit the execution of files through the kernel's fanotify API. The substantial difference is what the policy checks: in fact, in fapolicyd, it only checks the file path. For the source-to-execution chain, the proposed solution is most similar to Apple's Gatekeeper / notarization checks, which are run when files are executed. While these software, and the one proposed, can limit the execution of files, once they have been permitted to execute, other policy agents such as SELinux or AppArmor is still needed to police the behavior of running processes, as seen in 1.

VII. Conclusions

The solution proposed, for what the signature application layout and technique is concerned, is proven to be effective, adopting robust cryptographic algorithms, but also being agnostic of the underlying program nature. It successfully survives to files transfer and can be correctly enforced at runtime, being completely independent from the underlying FS, given that it does not need a physical file, but just a couple consisting of a payload accompanied by its signature data. It still relies on kernel APIs, though, which makes the solution heavily OS dependent, for what the enforcing is concerned.

When dealing with binaries only, provided that the OS user privilege separation is trusted, an interceptor leveraging on fanotify API is enough to achieve the desired result of being able to control whether or not a program can be run only based the trust of the signature it is shipping with.

Especially when it comes to interpreted runtimes, the enforcing mechanism shows its limits, leading to the need of a solution — *i.e.* an LSM-based implementation — that can gather more context information about processes, more robustness than the simple preliminary compromise of trusting the OS user privilege separation and more control to enforce more granular policies.

A. Future work

While the fanotify API we used was introduced for malware scanners, it is also limited in the information available, and currently the only way to get more would be to implement the checker as a kernel module / LSM. On a related note, interpreters can be modified to conduct security checks on their various forms of input, though this greatly expands the surface area of the setup. Additionally, the proposed solution leverages on the kernel's privilege separation to keep the public key database safe: currently, there is no other place to put it, due to the choice of cryptographic primitives, but a different choice may be able to more securely store keys in a TPM. Finally, while Apple's tightening of notarization checks may be a nuisance to power users, their work in the domain is certainly interesting and warrants a more in depth look.

References

[1] Inc., D. *What is a Container?* 2018. URL: https://www.docker.com/resources/what-container (visited on Apr. 20, 2020).

[2] De Lucia, M. *A Survey on Security Isolation of Virtualization, Containers, and Unikernels*. May 2017.

[3] Canada, E. R. *DigSig (Digital Signature... in the Kernel) and DSI (Distributed Security Infrastructure)*. 2009. URL: http://disec.sourceforge.net/.

[4] Singer, M. *Bsign*. 2002. URL: https://github.com/digsig-ng/bsign-mirror.

[5] Apple. *About System Integrity Protection on your Mac*. 2019. URL: https://support.apple.com/en-us/HT204899.

[6] Apple. *Safely open apps on your Mac*. 2019. URL: https://support.apple.com/en-us/HT202491.

[7] Odgaard, A. *macOS 10.15: Slow by Design*. 2020. URL: https://sigpipe.macromates.com/2020/macos-catalina-slow-by-design/.

[8] Linux. *fanotidy(7)*. 2019. URL: http://man7.org/linux/man-pages/man7/fanotify.7.html.

[9] Corbet, J. *The fanotify API*. 2009. URL: https://lwn.net/Articles/339399/.

[10] Redhat. *File Access Policy Daemon*. 2020. URL: https://github.com/linux-application-whitelisting/fapolicyd.

[11] Microsoft. *Integrity Policy Enforcement (IPE)*. 2020. URL: https://microsoft.github.io/ipe/.

[12] Dsafford et al. *Integrity Measurement Architecture (IMA)*. 2020. URL: https://sourceforge.net/p/linux-ima/wiki/Home/.

[13] Docker. *Content trust in Docker*. 2020. URL: https://docs.docker.com/engine/security/trust/content_trust/.

[14] Google. *Binary Authorization*. 2020. URL: https://cloud.google.com/binary-authorization.

[15] Latacora. *The PGP Problem*. 2019. URL: https://latacora.micro.blog/2019/07/16/the-pgp-problem.html.

[16] Unangst, T. *signify:Securing OpenBSD From Us To You*. 2015. URL: https://www.openbsd.org/papers/bsdcan-signify.html.

[17] Perez, B. *Seriously, stop using RSA*. 2019. URL: https://blog.trailofbits.com/2019/07/08/fuck-rsa/.

[18] Soatok. *Elliptic Curve Diffie-Hellman for Humans and Furries*. 2020. URL: https://soatok.blog/2020/04/26/a-furrys-guide-to-digital-signature-algorithms/.

[19] Josefsson, S. and Liusvaara, I. *Edwards-Curve Digital Signature Algorithm (EdDSA)*. 2017. URL: https://tools.ietf.org/html/rfc8032.

[20] Lee, M. *Backdoored Linux Mint, and the Perils of Checksums*. 2016. URL: https://micahflee.com/2016/02/backdoored-linux-mint-and-the-perils-of-checksums/.