A Trust Infrastructure for FreeBSD

Eric L. McCorkle

January 15, 2018

Abstract

This paper describes a public-key trust infrastructure for the FreeBSD operating system which manages the notion of delegated trust in the form of public-key cryptographic signatures. It is designed to integrate easily with existing userland applications as well as other similar components such as secure/trusted boot and the NetBSD VeriExec infrastructures, and to provide a workable foundation for future developments. Additionally, it is designed in such a way that existing post-quantum signature schemes can be used practically.

1 Overview

In the traditional UNIX operating system paradigm, trust is placed entirely in the kernel, which is expected to enforce the system's security model. The kernel in turn trusts users- the administrator in particular -to properly manage the parts of the system under their control. While this model is sufficient to maintain security properties, it is somewhat naïve and simplistic by more contemporary thinking on the topics of trust and security which seeks to account for such cases as malicious users, misconfigurations, and compromised systems or components in a more comprehensive manner.

In more modern developments, operating system security has increasingly moved in the direction of finer-grained access controls and toward the notion of *capabilities*. FreeBSD's Capsicum[10] framework is an example of such a framework. These developments indicate a move away from the blanket trust model of classical UNIX architecture towards a model where trust is managed and tracked at the granularity of specific operations. Additionally, cryptographic methods have been employed in the context of OS security. Secure boot mechanisms such as the UEFI secure boot standard[4] and OpenPOWER trusted boot[8] maintain a secure chain of custody through the boot process. The NetBSD VeriExec framework[6] uses a registry of message authentication codes (MACs) to verify executables during both boot and normal execution.

Finally, cryptography is an attractive tool for OS security, because it provides strong security guarantees without relying on a trust boundaries or trusted authorities. This provides strong tamper-resistance capabilities as well as the ability to maintain security guarantees over insecure communication channels and between system in an untrusted environment.

1.1 Public-Key Cryptographic Signatures and Trust

Public-key cryptographic signatures[9] provide a more nuanced model of trust. Public-key cryptography uses a *key-pair*, consisting of *public* and *private* keys, such that the private key cannot be effectively deduced from the public key. In public-key signatures, the private key is used to generate signatures and the public key is used to verify them. The typical use of this capability is similar to notarization of documents: an agent possessing the private key generates a signature for a given message, which can then be verified by anyone with the public key. Any alteration to the message will cause the signature check to fail, and the signature cannot effectively be forged without the private key.

An immediate use of this capability is to assign trust to some data object (which may represent an access token, an executable, configuration data, or many other things) under secure conditions, and then later to

verify that the trusted object has not been tampered with. A secondary use of this capability is to establish *chains of trust* by signing the public key of one keypair using another. This acts as an attestation that the signer *delegates* trust to the signee in some capacity. A third usage is to publish revocations of delegated trust, in the event that a key to which trust has been delegated is compromised.

This ability to delegate, manage, and verify trust has many potential applications in information security. This is currently used to great effect in application space; however, it has significant potential on the OS and kernel levels as well. In order to facilitate such applications, it is nesessary for the OS to possess a trust management infrastructure at the kernel level. This paper is dedicated to the design of such a system for the FreeBSD operating system.

1.2 A Public-Key Trust Infrastructure for FreeBSD

This paper proposes a design for an OS-level trust infrastructure for the FreeBSD operating system. This infrastructure is designed to be a flexible and powerful public-key trust management framework built into FreeBSD at the kernel level. It is designed to establish a notion of trust based on public-key signatures which can be made available to both kernel-space and userland services at runtime, thereby allowing the system to manage and delegate trust in a secure manner.

This system consists of two main components: the *runtime trust database* described in §2 manages trusted verification certificates in a running system, and the *trust base configuration* described in §3 establishes the trusted certificates when building systems as well as at boot-time. Additionally, there are several conventions for signing critical system components described in §4, including the *signed ELF* extension, which creates a convention for signing executable files. The trust infrastructure proposed in this paper has a number of possible uses, some of which are discussed in §5, and can integrate well with the NetBSD VeriExec framework[6] as discussed in §6. Finally, the design of the system is such that it can employ current post-quantum cryptographic signature schemes, as discussed in §7.

1.2.1 Design Goals

The following are the general design goals for the FreeBSD trust infrastructure:

- Observe and uphold the principles of open-source software- specifically the support of user control of their own systems
- Provide a general and extensible mechanism for managing trust in the form of public-key cryptographic signatures which can be used to build a wide range of new features
- At a minimum, support boot loader integration, signed kernel and modules, and signature checks on module loading at runtime
- Integrate with existing applications that use public-key infrastructure (PKI) with minimal or no modification required
- Provide functionality that can be adapted to a wide range of use cases
- To the greatest extent feasible, use existing components of the base system and avoid adding to it

2 Runtime Trust Database

The core of the trust infrastructure is the *runtime trust database*, which is a collection of verification certificates maintained by the kernel for the purpose of verifying signatures, thereby establishing the system's notion of trust at runtime. One or more *root trust certificates* form the foundational notion of trust for the system. These root certificates cannot be added, altered, or modified in a running system through ordinary

```
int trust_init(unsigned int nroots, cert_t *roots[]);
int trust_add_cert(cert_t *cert);
int trust_set_revlist(revlist_t *revlist);
int trust_check_sig(const sig_t *sig, unsigned int nchain, const cert_t *chain);
```

Figure 1: Core In-Kernel Trust Database API

means.[1] In order to prevent a failure scenario where all root certificates expire, leaving a system without a viable notion of trust, root certificates are checked for expiration *only* when established as such and remain valid thereafter until the system is rebooted. *Intermediate trust certificates* reperesent a notion of *delegated trust*. As such, they can be added and revoked at runtime, and they expire automatically. Intermediate certificates can also delegate trust to other intermediate certificates; however, every intermediate certificate must have a valid trust chain back to a root certificate.

On systems without special hardware, the runtime trust database is implemented as a simple in-memory forest data structure, with root nodes representing root trust certificates and edges to intermediate certificates from their signer,[2] and a hash index to allow fast lookups of a specific key. If a hardware platform provides a reliable hardware key-store mechanism, that can be used instead of the in-memory database to store keys more securely.

2.1 In-Kernel Interface

The in-kernel interface for the runtime trust database is fairly simple. Figure 1 shows the core functionality for the API. The root certificates are provided at initialization time using `trust_init`, and are fixed thereafter. Intermediate certificates are added using `trust_add_cert`, which checks that the new certificate is signed by an existing root or intermediate certificate, and that the new certificate is not present in its signer's revocation list. Each root and intermediate certificate has a revocation list associated to it. This list is empty by default, and can be set using `trust_update_revlist`.[3] When a revocation list is updated, any child certificates which are revoked are immediately canceled along with all of their descendents. Certificates are checked using `trust_check_sig`, which requires that the signature (or the root of the certificate chain argument, if provided) be signed by a valid root or intermediate key.

The portion of the API not shown here provides additional management functionality, such as checking intermediate certificates for expiration and enumerating all of or some subset of the trusted certificates.

2.2 Trust `devfs` Interface

The primary interface to the FreeBSD trust system utilizes the device filesystem (`devfs`) to present a number of device nodes. It has three primary functions: registering new intermediate trust certificates, revoking intermediate trust certificates, and reading certificates back. The `devfs` interface is presented in order to allow existing applications to interact with the trust infrastructure without modification, and to allow administration to take place without specialized control programs.

Trust system device nodes reside under the `trust` directory in the `devfs` filesystem (`/dev/trust` on a typical system). The `trustctl` device node is a write-only device node to which either X.509 certificates or revocation lists in the binary DER[4] encoding can be written. Writing an X.509 certificate which is signed by an existing root or intermediate trust certificate *and* which is not present in the signer's revocation list will

[1] It is possible that the root certificate set could be altered at runtime in a system using a hardware security mechanism such as a TPM; however, the specifics of such a mechanism are beyond the scope of this paper.

[2] In a PKI-style system, this is indeed a forest, as all certificates have a single signer. In a PGP-style system, this would be a general graph.

[3] X.509 revocation lists are signed by the certificate to which they apply, thus they can be associated with a single certificate.

[4] The binary DER encoding is chosen as the input format because it is easy to parse, hopefully minimizing the probability of a parsing bug or vulnerability.

result in the new certificate being installed as an intermediate trust certificate. If a valid X.509 revocation list signed by an existing trusted certificate is written to `trustctl`, then it is installed as the revocation list for its signer, and any intermediate trust certificates in the revocation list will be immediately removed from the runtime trust database, along with all their descendent certificates.

Certificates can be read back in bulk from two read-only device nodes: `certs` and `rootcerts`. The `certs` node outputs all trust certificates, while the `rootcerts` node outputs only the root certificates. Both nodes output certificates as a concatenated sequence of PEM-encoded X.509 certificates, which is the standard format used by many applications to store their CA certificates. This is intentional, with the purpose of allowing these applications to directly use the runtime trust database as the source for their own root certificates without any modification necessary.

2.3 Obtaining Root Keys

In the kernel API described in §2.1, the trust system must be initialized with the root keys, which cannot be changed thereafter. There are several ways by which the kernel and the loader can obtain the root keys, each of which has relative advantages and disadvantages.

The first method is for the keys to be statically compiled in to the kernel and the loader. Barring a TPM-like hardware solution or the ability to obtain keys from a previous stage, this is the only method for the loader which maintains a secure chain of custody, provided that the loader itself is signed and verified by the previous stage or is installed directly to firmware through a mechanism like coreboot. Kernels with built-in root keys are similarly secure, provided they are verified by the loader.

An alternative is to have the kernel obtain its root keys through the `keybuf` mechanism. This mechanism was added to enhance GELI support, and uses the kernel boot-time metadata framework to send keys from the loader to the kernel in a flexible and extensible manner. It is straightforward to add new key types to the `keybuf` interface to represent the incoming root keys. The kernel then retrieves these keys at boot time in order to obtain the root keys.

Both options can be used, and the two can be combined. Highly secure systems will likely want to disable the `keybuf` mechainsm and build keys in directly. The `keybuf` option, on the other hand, in ideal for standard builds, as it allows one kernel image to be used on many different systems with their own root keys. A combination of the two allows root keys to be built in, but more to be added if need be.

3 Trust Base Configuration

The runtime trust database establishes the notion of trust in a *running* system; however, since many FreeBSD systems are built from source locally, it is also necessary to establish notions of trust at build time. Furthermore, it is useful to provide a way to automatically set up intermediate trust certificates at boot-time. The *trust base configuration* accomplishes both tasks.

The trust base configuration consists of a collection of certificates and private keys stored at `/etc/trust`. Both root and intermediate certificates are stored in this configuration, but their handling differs. Root trust certificates are stored under `/etc/trust/root`, with certificates at `/etc/trust/root/certs` and keys at `/etc/trust/root/private`. The root certificates are converted to binary data in a linkable format during the system build and linked into the loader and possibly the kernel. The root keys are present only for signing purposes, and are not built into the loader or kernel (as doing so would defeat the purpose of public-key cryptography). Intermediate certificates are stored at `/etc/trust/certs`, and are expected to be signed by a root key, or by another intermediate certificate. These certificates are loaded as intermediate trust certificates at boot-time using the `devfs` interface described in §2.2. As with root keys, intermediate keys are present only for signing.

It may be the case in some arrangements that a system possesses a root or intermediate certificate with no corresponding key. Such certificates are known as *third-party* certificates. Certificates that do have a corresponding private key are known as *local* certificates. Third-party certificates are used to grant trust to some external party, such as a vendor or an institutional build system.

3.1 Possible Configurations

There are many possible ways a trust base configuration can be set up, each of which has different advantages. We describe three common cases here, but this treatment is by no means exhaustive.

3.1.1 Preferred Configuration

The preferred trust base configuration has a single system-specific local certificate and key which represent the owner of the system. All additional trust certificates are signed by this system-specific key and are treated as intermediate certificates. The system-specific key is generated during system installation, and is never exported from the system.

This configuration is preferred because it gives full autonomy and control to the user. The only root key (which cannot be revoked at runtime) is the system key, which is created by the owner of the system and remains under their control. All other trust is granted by this key, and can be revoked at runtime if the user wishes. For example, if a user wishes to trust binaries signed by the FreeBSD Foundation, they would simply install the Foundation's third-party certificate as an intermediate certificate. This way, a user is free to trust whomever they wish, but is not *required* to do so. This configuration is appropriate for most systems and nearly all individual users.

3.1.2 High-Security Institutions

High-security institutions may wish to maintain tighter control on their systems, requiring that all installed programs be distributed from a tightly-controlled set of machines. Additionally, such institutions will likely want to maintain pre-built machine images which can be rapidly deployed to production.

In such a case, it is inappropriate to maintain a local copy of the signing key on each machine. For one, this would allow machines to circumvent the central build system. Moreover, it would allow attackers to potentially capture the key and sign malicious programs. Therefore, in such an environment, we maintain an institution-wide key-pair and deploy only the certificate to each machine (aside from the build machines) as the root trust certificate. This way, the trust configuration remains centrally-controlled and the signing keys are not at risk of being compromised.

3.1.3 Standard FreeBSD Images

The FreeBSD Foundation publishes readymade system images as part of every release, often in the form of bootable ISO or memstick images. If these are to ship with an active trust system, then it is necessary to establish a common trust root for all such images. This can be done in essentially the same manner as the high-security institution described in §3.1.2, with the FreeBSD Foundation maintaining its trust root key.

This setup has the critical problem that it provides the user with no control over their own system. Thus, it should be considered acceptable *only* for bootable media images and *not* for installed systems. The FreeBSD installer should set up a preferred configuration as described in §3.1.1 as part of the installation process.

4 Signed System Components

One of the most immediate uses of the trust system is to provide tamper-resilience by allowing the boot loader and the kernel to verify signatures on critical system components. The most crucial by far would be the kernel itself and its loadable modules; however, there are many possibilities that extend beyond this use case. A critical part of providing this functionality, however, is designing a workable convention for signing critical system components. We cover these conventions in this section.

4.1 The Signed ELF Binary Extension

One of the key features of the trust system is the ability to certify executables and libraries. This can be used for a variety of purposes, ranging from kernel and module signature verification, to privileged executables, and even the requirement that all executables be signed in extreme security cases.

The signed ELF binary extension makes use of features of the ELF binary format[3] to attach signatures as a kind of metadata. The format is specifically designed such that a signed executable can be crafted and verified using the `objcopy` and `openssl` command-line utilities.

A signed ELF binary contains an extra metadata section, named `.sign`, which contains a detached signature in the Cryptographic Message Syntax (CMS) format. It is expected that this detached signature will omit extra metadata such as certificates, S/MIME capabilities, and other metadata, resulting in a reasonably compact signature.[5] The signature itself is computed against the entire contents of the ELF binary, but with the `.sign` replaced with all-zero data of equal length.

This signature-calculation procedure is more complex to calculate than it is to verify. Calculating a signature involves adding the `.sign` section to the ELF data, calculating the signature, then writing it into the section. By contrast, the signature can be verified simply by copying out the signature data, overwriting the `.sign` section with zero data, then perfoming signature verification. This is by design; binaries are likely to be signed as part of a build process, which is not time-critical. However, they will be verified repeatedly as part of program and library loading, which *is* time-sensitive.

4.2 The `signelf` Utility

Signed ELF binaries can be created using only the `objcopy` and `openssl` command-line utilities. While this is excellent for portability, the procedure is somewhat tedious and it lacks certain desirable features like batch-signing and ephemeral keys. To address these shortcomings, the FreeBSD trust infrastructure includes a command-line utility- `signelf` -which is designed for streamlined batch-signing/verification of large numbers of executables.

Moreover, `signelf` is able to generate an ephemeral signing key for a batch of signatures, then write out the public-key certificate used to verify the signatures. The ephemeral private key is destroyed at the end of the procedure. This has additional security advantages in that once the batch-signature is complete, no one can ever produce signatures with the same key again.

4.3 Signing Critical Configuration Files

The signed ELF format provides a way to sign and verify executables and libraries; however, the FreeBSD operating system also contains a number of textual configuration files which an operator will want to protect from tampering. The clearest example of this are the loader configuration files such as `loader.conf`, `loader.4th`, and others. These files control the actions of the loader, and thus can be used to circumvent any sort of secure boot process if they are tampered with.

In the case of loader configuration files, there are few enough of them to warrant a separate strategy: namely the use of Cryptographic Message Syntax envelopes (i.e. *attached* signatures). The FreeBSD loader first checks for files with the `.cms` extension (for example, `loader.conf.cms`), which it expects to contain a CMS envelope. If the file is found, it checks the signature and extracts the contents. If the envelope file is not found or the signature check fails, it reports an insecure boot.

Outside the scope of the boot loader, there are more options as well as more variations on the kind of critical data. In general, it is not a good idea to attempt a one-size-fits-all solution here; rather, we should provide a toolbox of techniques for use. CMS envelopes are good at protecting files in a way that minimizes administrative overhead, but they also require modification of the applications. Detached CMS signatures stored in a separate file represent an alternative to this, but complicate administration. In some cases, configuration files may be able to store or point to signatures for critical elements. Lastly, some file formats

[5] For RSA-4096 signatures, the detached signature is only about 800 bytes in size

may support arbitrary metadata the way ELF does. These alternatives should be considered and weighed against eachother when designing a solution for critical data.

5 Possible Uses

The FreeBSD trust system is designed to be a highly general system with the intent of empowering further developments in the FreeBSD system and in operating systems generally. It has many possible uses, some of which we elaborate here.

5.1 Secure Boot and Kernel

The most direct use of the trust system is to maintain a secure chain of custody from the boot loader to the kernel at runtime, and to ensure that only authorized kernel modules are loaded. This is accomplished by having the boot loader check signatures on the kernel as well as on the critical files.

The loader can obtain its trust root certificates in one of two ways: either the certificates can be built into the loader, or it can obtain the certificates from a boot service such as EFI. In truth, these mechanisms are equivalent; either way, the loader must be signed, and it is assumed that the signature is checked by whatever firmware loads it. It is likely preferable, however, that the key be built into the loader, as this will likely provide access to a better set of ciphers.

5.2 System Root Certificate Management

Another use of the trust system is to centralize certificate management for the entire OS. Most user programs manage their root certificates by having their configuration files point to PEM-encoded certificates which act as the root certificates. While this is workable, it leaves the management of root certificates up to each application. The trust system provides a more certralized mechanism, effectively binding the applications' notions of trust to the system's. Indeed, the `devfs` interface for reading back certificates described in §2.2 is designed to allow applications to use the runtime trust database in lieu of an application-specific CA configuration.

5.3 Signed Executables and Libraries

The signed ELF standard can also be used to sign regular executable files. This has a wide variety of uses. One possible use is to require valid signatures on `setuid` binaries, preventing attackers from tampering with privilege-escalating programs. On high-security systems, it may be desirable to require *all* executables be signed, thereby preventing any tampering. There are many other possibilities, which are far too numerous to list exhaustively here.

5.4 Delegated Capabilities

One of the more interesting possibilities for the trust infrastructure lies in its relationship to the Capsicum capabilities framework[10]. In Capsicum as it exists now, capabilities are granted on a particular machine, and they are only valid on that machine. The trust system provides a mechanism whereby trust can be delegated to other machines, providing the groundwork for remote-issued capabilities. In such a system, remote capabilities take the form of a signed message which is checked against the local trust database. If the capabilities were issued by a trusted machine, then the signature check will succeed, and the capabilities will be considered valid. The potential uses for such a delegated capability system are far-reaching and beyond the scope of this paper.

5.5 Kernel-Side Key Agreement

Key agreement protocols are another public-key cryptographic protocol by which counterparties exchange public keys, and thereafter are able to derive the same secret value, which is typically then used in symmetric-key cryptography to secure a session. Key agreement establishes *confidentiality*; it does not establish *trust*, and protocols typically must rely on an additional signature verification step to avoid middleman and other forgery-based attacks. This is one of the reasons why infrastructures such as PKI are necessary.

Key agreement is typically done in user space; however, it can potentially serve a purpose in kernel space as well, particularly in the context of a delegated credential system as discussed in §5.4. The trust infrastructure provides a vital function for in-kernel key agreement: namely the ability to verify that the public keys involved have not been tampered with by a potential attacker.

6 Integration with VeriExec

The NetBSD VeriExec framework provides the ability to register a manifest file containing MACs (message authentication codes) for various executables and files in the system. These are then checked against the manifest when loaded, and generate an error if the authentication code check fails. At first, this may seem to be an alternative to the trust framework described here; however, it is not and in fact, the two systems can be made to interact quite well.

6.1 Tradeoffs

The key advantage of the VeriExec system is that it stores MAC information out-of-band from the files to which it applies, and therefore requires no modification of the files. Additionally, MAC codes are typically smaller and faster to check than public-key cryptography by an order of magnitude. They are also secure against quantum attacks, unlike most public-key cryptography in common use.

The key downside is that the manifest itself becomes an attack vector. Additionally, as MAC codes are a symmetric cryptographic method, they cannot provide the trust delegation capabilities of a public-key system. Finally, the manifests must be updated every time the trusted components are altered in any way, and it is not safe to allow manifests to be loaded or altered during normal runtime. A less serious downside is that VeriExec manifests currently associate a MAC with a path as opposed to specific contents, which somewhat limits the flexibility of system administration.

6.2 Integration

The most obvious method for integrating the trust system and the VeriExec system is to require all manifests to be signed by a trusted key. This opens the possibility for manifests to be added and updated dynamically at runtime and for manifests to be generated on a remote machine and trusted locally.

The VeriExec system can be further combined with the signed ELF standard in a way that provides the same flexibility as signed ELF files, but uses MAC verification. In this scheme, executables are marked with a UUID (presumably in a `.uuid` section). These UUIDs are used in lieu of paths in a VeriExec manifest to associate a MAC with the executable image. This scheme has the advantage that a single UUID can act as a proxy for an arbitrary number of signatures,[6] and that MAC checks against a UUID-bearing executable do not require zeroing out a section in order to compute.

Under such a scheme, most executables would be indexed by UUIDs and associated with a MAC through the VeriExec manifest (which is in turn verified by the trust system). However, the signed ELF format still allows executables to be signed and trusted without having to update the MAC registry, thereby maintaining the flexibility provided by signed ELF binaries.

[6] In fact, the compiler toolchain can potentially be modified to generate UUIDs for every executable and shared object using a weaker hash function like SHA-1 or RIPEMD-128

7 Post-Quantum Cryptography

Quantum computation[7] has significant implications for public-key cryptographic systems. At the time of writing, it is a reasonable belief that quantum computers capable of attacking existing public-key cryptosystems will exist in a timescale of more than 5 years and less than 50. This has profound implications for public-key cryptography, and efforts are underway to develop quantum-resistant public-key schemes (known as post-quantum crypto). For the most part, quantum computing affects only public-key cryptography; symmetric-key methods are subject only to a theoretical attack.[7] Quantum-resistant signature schemes have been developed, but at this point they are either too new and poorly-understood to be trusted, or else their signature sizes are too large to serve as a drop-in replacement.

The trust system is under considerably less pressure to deploy quantum-resistant crypto, as it is a signature-checking system used for real-time authorization. Whereas protocols such as TLS can be recorded and attacked at a later date in order to obtain secret information, forging signatures for old keys in the trust system accomplishes nothing if the system administrator has since moved to quantum-resistant signatures and revoked the old certificates. Thus, the trust system can adapt to the advent of quantum computing simply by switching to quantum-resistant algorithms when necessary.

Even so, the usage patterns of the trust system are such that it is possible to make practical use of existing post-quantum signature schemes. This section describes such cases.

7.1 Hash-Based Signature Schemes

Hash-based signature schemes provide information-theoretic security, but at the cost of larger signatures or a non-standard (and brittle) interface. Stateful hash-based signature schemes such as XMSS[5, 2] require a "state" which is updated every use, and accidental re-use of an old state destroys the security properties of the scheme. In practice, such a scheme is unsafe for general use. The SPHINCS signature scheme[1] is stateless, but has signatures 40Kib in size: a size which is too large to be practical for use in the signed ELF format, or for signing text files. However, the trust system can effectively make use of both schemes.

7.1.1 Safe Use of Stateful Hash Signatures

Stateful hash signature schemes such as XMSS break from the classic definition of signature schemes by including a "state", which is updated on every signing operation. Only one message can be safely signed for a given state; signing multiple messages breaks the security properties and potentially allows an attacker to forge messages. As such, this is extremely dangerous to use in the same way as traditional signing keys, *especially* in the presence of features like ZFS snapshots.

There are, however, two "safe" uses of stateful signatures, both of which produce an ephemeral key only for a limited use. The first is for "batch-signing" a collection of messages. In such a case, a key is generated, used to sign a batch of messages, then the private portion and ending state are discarded, ensuring that no repeat signings occur. This kind of signing with ephemeral keys is already implemented by the `signelf` utility discussed in §4.2, and the utility could easily be extended to use XMSS or a similar scheme.

In the second, the kernel maintains the private key and state as a resource to which it controls all access, and which it never allows to be persisted. In this pattern, the key is generated at startup, installed as a trust root, and destroyed at shutdown, with the kernel managing the private key and state to prevent repeat signings. Such a system can be used to issue delegated credentials that expire whenever the system goes offline, and could form the basis of a delegated credential system like the one suggested in §5.4.

[7] Symmetric-key ciphers and hashes are subject to the quadratic-speed Grover iteration, which effectively halves their security factors as opposed to the more catastrophic hidden-subgroup attacks that offer an exponential speedup against public-key cryptography. However, a Grover iteration attack against even a 128-bit key would require a very large quantum memory and a quantum computer capable of remaining stable for very long periods of time.

7.1.2 Practical Use of SPHINCS Signatures

Because of the 40Kib size of SPHINCS signatures, it is not practical to use them interchangeably with RSA or ECC signatures. In order to be practical, we must use them only where signed objects are either few in number or large enough that the signatures don't represent a significant portion of the object. Fortunately, both of these apply to likely uses of the trust system. A VeriExec manifest, for example, is likely to contain large numbers of MAC codes for numerous system files. This file can easily dwarf a SPHINCS signature, and even if it does not, that there are likely at most a handful of manifests per system and the signatures are typically checked only once during a system's uptime makes SPHINCS signatures practical here.

8 Conclusion

The trust system proposed herein is a flexible and powerful infrastructure for establishing a notion of trust for the FreeBSD kernel and operating system. Trust is represented using public-key signatures and managed by an in-kernel database of public-key certificates and revocation lists. The trust system is presented to userland through a `devfs` interface which can be used directly as in the configurations of many applications which use PKI-style certificates. The proposed system also includes standards for signing ELF binaries and critical configuration files, and can integrate with the existing NetBSD VeriExec framework in beneficial ways. The trust system is designed to be flexible and general, and can be adapted to a variety of configurations and scenarios depending on the needs of the situation. Finally, the trust system's design allows existing post-quantum cryptographic methods to be utilized in a practical manner, thereby providing security guarantees that withstand the advent of scalable quantum computing. With these qualities, the trust system stands to provide a general and powerful foundation for the development of new OS security features going forward.

References

[1] Daniel J. Bernstein, Daira Hopwood, Andreas Hlsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-O'Hearn. SPHINCS: Practical stateless hash-based signatures. Cryptology ePrint Archive, Report 2014/795, 2014.

[2] Johannes Buchmann, Erik Dahmen, and Andreas Hülsing. XMSS - A practical forward secure signature scheme based on minimal security assumptions. In *Proceedings of the 4th International Conference on Post-Quantum Cryptography*, PQCrypto'11, pages 117–129, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] TIS Committee. Executable and linking format (ELF) specification. Specification, Tool Interface Standard TIS, May 1995.

[4] UEFI Committee. Unified extensible firmware interface specification. Specification, Unified EFI Inc., May 2017.

[5] Andreas Huelsing, Denis Butin, Stefan-Lukas Gazdag, Joost Rijneveld, and Aziz Mohaisen. XMSS: Extended hash-based signatures. Internet-Draft draft-irtf-cfrg-xmss-hash-based-signatures-12, Internet Engineering Task Force, January 2018. Work in Progress.

[6] Brett Lymn. VeriExec framework. Component of NetBSD Operating System.

[7] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[8] Dean Sanner. Secure and trusted boot for OpenPOWER. In *2016 OpenPOWER Summit*, 2016.

[9] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley-India, 2007.

[10] Robert Watson and Johnathan Anderson. Capsicum framework. Component of FreeBSD Operating System.