# Start your ENGINEs: Dynamically Loadable Contemporary Crypto

# Start your ENGINEs: dynamically loadable contemporary crypto

Nicola Tuveri
Tampere University, Tampere, Finland

Billy Bob Brumley
Tampere University, Tampere, Finland

***Abstract*—Software ever-increasingly relies on building blocks implemented by security libraries, which provide access to evolving standards, protocols, and cryptographic primitives. These libraries are often subject to complex development models and long decision-making processes, which limit the ability of contributors to participate in the development process, hinder the deployment of scientific results and pose challenges for OS maintainers. In this paper, focusing on OpenSSL as a de-facto standard, we analyze these limits, their impact on the security of modern systems, and their significance for researchers. We propose the OpenSSL `ENGINE API` as a tool in a framework to overcome these limits, describing how it fits in the OpenSSL architecture, its features, and a technical review of its internals. We evaluate our methodology by instantiating `libsuola`, a new `ENGINE` providing support for emerging cryptographic standards such as X25519 and Ed25519 for currently deployed versions of OpenSSL, performing benchmarks to demonstrate the viability and benefits. The results confirm that the `ENGINE API` offers (1) an ideal architecture to address wide-ranging security concerns; (2) a valuable tool to enhance future research by easing testing and facilitating the dissemination of novel results in real-world systems; and (3) a means to bridge the gaps between research results and currently deployed systems.**

## I. Introduction

Following current common best practices for the development of secure systems, most applications rely on cryptographic primitives for which widely accepted standards have been published after extensive studies to ensure that breaking them is believed computationally infeasible. As the available computation power increases and new attacks and algorithms to improve the performance of theoretic attacks are developed, these standards are periodically revised to raise the security level of the recommended primitives.

Considering the complexity of protocols and algorithms and the number of potential pitfalls concerning details at every level of abstraction, the software implementation of such evolving standards is a daunting and complex task and application developers are strongly advised not to implement their own crypto but to rely on existing well-established cryptographic libraries, among which OpenSSL[1] is the most widely adopted. Still, in addition to the complexity of their programming interfaces, bugs and defects in these libraries are often the cause of cryptographic failures in the security layer of modern information systems.

Previous research focused on the development of new cryptographic software libraries as a solution to these problems: with respect to this paper, the NaCl [1] project—and its fork `libsodium`[2]—are especially relevant, as they aim at providing practical and efficient strong confidentiality and integrity with a special emphasis on the simplicity of the provided programming interface. Unfortunately, after years, we notice that, even if gathering momentum and being adopted in various new projects, this approach fails in meeting the needs of mainstream projects.

Another related development sprouting off this research is HACL*[3] [2], a verified cryptographic library that implements the NaCl API: it provides mathematical guarantees on memory safety, timing side-channel safety and functional correctness (with respect to the published primitive specification), while producing portable C code that is as fast as state-of-the-art C implementations.

Alongside the development of NaCl, research efforts were also spent on the related SUPERCOP benchmarking suite as part of the eBACS project [3] for the benchmarking of cryptographic systems: many researchers from industry and academia submitted new cryptographic primitives or alternative implementations for benchmarking. Included submissions are benchmarked across different architectures and toolchains and undergo several automated tests. But still, most of these programming and scientific efforts fail to reach widespread adoption through mainstream libraries. We defer to Section II for an in-depth analysis of the limits of the OpenSSL project from the point of view of researchers that might help explaining this gap. In here, to further substantiate the existence and relevance of this gap, we focus on the implementation of emerging cryptographic standards based on Curve25519 [4, 5]. X25519, the Diffie-Hellman cryptosystem, originally released in 2005, promises, due to the properties of the underlying curve design, simpler and faster implementations, with enhanced resistance to side-channel attacks. Ed25519, formally introduced in 2011, is a digital signature system based on the twisted Edwards equivalent of Curve25519. It delivers fast digital signature generation and verification, short signatures, and enhanced resistance to side-channel attacks.

These cryptosystems have since gathered momentum, gaining official support in OpenSSH in 2014, in BoringSSL in 2015, in the Google Chrome Internet browser TLS/QUIC protocol support in April 2016, and finally becoming part

[1] https://www.openssl.org

[2] https://libsodium.org/

[3] https://github.com/mitls/hacl-star

of IETF RFC 7748 [6] (X25519) in January 2016 and RFC 8032 [7] (Ed25519) in January 2017, and becoming officially recommended for implementations of TLS 1.3 (RFC 8446 [8]) and earlier versions (RFC 8422 [9]) since August 2018.

OpenSSL officially added support for X25519 in August 2016, with release 1.1.0, while Ed25519 support landed only very recently in a release version, after a long development cycle, with the release of 1.1.1 on September 11, 2018.

While implementation details are examined in Section III-D, here we highlight that the original implementation chosen by the OpenSSL development team favored portability over optimization, and as a result using these cryptosystems in applications built on top of OpenSSL, do not always yield the expected performance in comparison with other elliptic curve (EC) implementations (e.g., NIST P-256).

This does not seem to be a consequence of a lack of trusted optimized implementations for the most widespread architectures supported by OpenSSL, as the mentioned SUPERCOP project includes several implementations tested on different architectures and alternative libraries like `libsodium` and BoringSSL ship optimized implementations for popular architectures. It rather shows that new research, even after the time required for gaining trust and widespread adoption in other mainstream projects and standards, still requires a long additional time to be included in OpenSSL, affecting researchers, developers and users.

*Our contribution:* To address these limits, in this work: (1) we present an analysis of the OpenSSL `ENGINE API` and its benefits for bridging gaps between cryptographic research and practical real-world implementations of cryptosystems—the analysis in itself is novel, as available resources appear to be outdated; (2) we develop an `ENGINE` to demonstrate how to use the `ENGINE API` as a framework to transparently integrate alternative implementations or new functionality in OpenSSL, making them available to existing applications; (3) we evaluate experimental results, by benchmarking our `libsuola ENGINE`, demonstrating the viability and the benefits of the proposed solution across different versions of OpenSSL.

As discussed later in Section III, the entire existence of the OpenSSL `ENGINE API` is to provide alternative implementations; our novelty instead lies in our "shallow" engine concept, bridging APIs of existing libraries to seamlessly realize this functionality and allowing easy selection of several different backend providers for it. Even though performance gains are a nice side-effect, the main values of using the proposed framework come from (1) the integration of missing functionality in end-of-life, yet still widely deployed, versions of OpenSSL; (2) transparent access to the integrated functionality to existing applications, requiring exclusively changes to system configuration; (3) freedom of choosing alternative implementations for new or existing functionality (e.g. choosing a formally verified implementation like HACL*); (4) ease of testing and benchmarking in real-world scenarios and dissemination to a larger user audience of novel implementations and primitives for researchers.

*Outline:* Section II presents further claims motivating our contribution. Section III contains an analysis of the OpenSSL architecture and the `ENGINE API`. Section IV presents `libsuola`, an `ENGINE` we developed to demonstrate how to use the `ENGINE API` as a framework to offer alternative or missing implementations in OpenSSL. In Section V we present and evaluate experimental results comparing the default implementation of X25519 and Ed25519 (or the lack thereof) against the implementations provided through our custom `ENGINE` and analyze the limits of our proposed methodology and how it addresses the concerns exposed in Section II. Section VI presents a brief analysis of related work, focusing on the mechanisms deployed by other security libraries, frameworks and operating systems to allow the transparent adoption of alternative cryptographic implementations. We conclude in Section VII.

## II. Motivation

As mentioned in Section I, this work is partially motivated by the rigidity of currently deployed, ubiquitous cryptography software libraries. This rigidity impacts real-world security at least via two avenues, which this section summarizes: it amplifies software assurance issues due to the small circle of contributors, and furthermore restricts the practical ability of OS vendors to provide predictable and steady support throughout the product life cycle.

While our intent is not to belittle the efforts of the OpenSSL project, to motivate our contributions, here we briefly highlight some limits of the project from the point of view of researchers that might explain the gap highlighted in the introduction: (1) lack of unified quality reference documentation on the overall software architecture of the library, on API design choices, and frequently on the details of public and internal functions and data structures; (2) complex ad-hoc build system; (3) strong constraints on the choice of programming languages (C and custom "augmented" ASM syntax) and coding style; (4) being a complex and huge project ongoing very active development, it implicates higher maintenance costs for external developers to keep their submissions up--to-date during the contribution process, which in turn can become quite long due to the double review constraint (see e.g. [10] which, while contributed in early 2009, did not get mainlined until late 2011, finally reaching a release version in early 2012); (5) contributing a new feature usually requires splitting the contributed code in smaller units to facilitate and speed up the review process and collect feedback and consensus on development choices for the subsequent units. In turn, generally, this might slow down the overall development process and increase the maintenance costs for contributing developers; (6) inclusion of new implementations or features usually undergoes a long decision-making process rarely compatible with research timelines; (7) trust, quality assurance, and IPR issues force the core development team to be very conservative in the decision-making process, thus even in cases where time is a minor issue, the final result can still be a rejection. In these cases, it is up to the researchers to maintain a custom set of patches and documentation to make their contribution available to users and other researchers for further research (see e.g. [11] where the proposed cipher suite is no longer compatible with the newer OpenSSL API).

### A. Software assurance

The OpenSSL codebase is maintained by a small team of developers, mostly on a voluntary basis, and few individuals working full-time on the project. This practically restricts the ability of contributors and end users to, e.g., control what alternative cryptographic primitive implementations are featured in the library. Upkeep of end user custom builds is costly and does not scale well, hence the logical choice is to stick to the version and feature set provided by the OS vendor, overwhelmingly driven by the choices of OpenSSL core developers themselves. This is a poor model for security-critical software: it leads to e.g. limited accountability in the decision-making process, and lack of assurance that the included code is functionally correct. In what follows, we give some examples of extremely security-critical code paths in OpenSSL that lack software assurance (see e.g. [12] for a broader survey). We strongly reiterate here that our work should not be construed as diminishing the contributions of the OpenSSL project—which at RWC'18 was awardedwith the Levchin prize "for dramatic improvements to the code quality"—but rather we use these observations to motivate our research and later demonstrate how our framework can alleviate this security burden, shifting it back towards developers and cryptographers.

*Software defects:* Biham et al. introduced the concept of bug attacks in 2008 [13], highlighting the importance of cryptography implementation correctness to protect private keys. In 2011, Brumley et al. presented the first (and only, as far as we are aware) real-world bug attack [14], remotely recovering P-256 private keys from a TLS server by exploiting a carry propagation software defect in OpenSSL 0.9.8g finite field arithmetic (CVE-2011-4354). The defect (and vulnerability) resurfaced later in the Nettle cryptography library [15, Sec. 3.1]. CVE-2014-3570 discloses a defect in multi-precision integer squaring, potentially affecting RSA, DSA, and ECDH cryptosystems in OpenSSL 0.9.8+. CVE-2016-7055 discloses a carry propagation bug leading to incorrect Montgomery multiplication results for 256-bit inputs, affecting ECDH cryptosystems for Brainpool P-512 elliptic curve in OpenSSL 1.0.2+. CVE-2017-3736 discloses a carry propagation bug leading to incorrect Montgomery squaring results, affecting RSA, DSA, and DH cryptosystems in OpenSSL 1.0.2+; CVE-2017-3732 and CVE-2015-3193 are similar but distinct defects. CVE-2017-3738 discloses an overflow bug in Montgomery arithmetic for 1024-bit moduli leading to incorrect multiplication results, affecting RSA, DSA, and DH cryptosystems in OpenSSL 1.0.2+. To summarize, these security vulnerability disclosures unfortunately document a track record of functional correctness failures in OpenSSL, putting the security of private keys at risk. Our proposed methodology allows any developer to provide alternative implementations of cryptosystems decoupled from the OpenSSL codebase with higher functional correctness assurance—e.g., formally verified libraries [2, 16]. We demonstrate this by providing an option to select HACL* [2] as the backend provider of the X25519 and Ed25519 implementations.

*Side-channel security:* In 2004, Percival discovered [17] the first cache-timing attack on public key cryptography in OpenSSL, recovering RSA keys by exploiting key-dependent table lookups in sliding window exponentiation (CVE-2005-0109). OpenSSL responded by (1) implementing (against expert advice to avoid cache line-level lookups [18, Sec. 15]) the scatter-gather technique [19] during exponentiation to reduce the amount of leakage; (2) adding a runtime flag that controls whether or not to follow this secure code path. Scatter-gather is not a full mitigation but a hedge [20], and in 2016 Yarom et al. implemented an attack utilizing cache-bank conflicts [21]. OpenSSL responded (CVE-2016-0702) by tweaking scatter-gather parameters to further reduce the amount of leakage, hence the root cause of the vulnerability is still present in OpenSSL as of this writing. Also in 2016, Pereida García et al. discovered a defect in the way OpenSSL handles the runtime flag added in 2005 and used it to recover DSA private keys from TLS and SSH servers [22], i.e. the fix to the 2005 vulnerability was in fact not being activated; a bug present for over a decade (CVE-2016-2178).

In 2009, Brumley and Hakala discovered the first cache-timing attack on ECC in OpenSSL, recovering private keys from scalar multiplication [23]. OpenSSL responded by implementing constant-time paths on certain architectures for elliptic curves P-224, P-256, and P-521 based initially on Käsper's work [24] and later a P-256 contribution from Intel [25]. The OpenSSL team declined contributed patches to blanketly mitigate the vulnerability [26]. Leaving the vulnerability present for all other curves, in 2014 Benger et al. improved the 2009 result and targeted the 256-bit BitCoin curve [27]. In 2015, van de Pol et al. further reduced the number of required signatures to recover private keys [28], and Allan et al. even further in 2016 [29]—all from the initial code path shown vulnerable in 2009 but only recently fixed [30] due to CVE-2018-5407 [31].

To summarize, unfortunately OpenSSL has a dubious track record regarding side-channel security: from stopgap countermeasures that do not stand the test of time, to implemented mitigations that are never activated due to defects, to no response at all when code paths are shown to be vulnerable. Our proposed methodology allows any developer to trump these OpenSSL design decisions and provide side-channel secure implementations largely independent from the OpenSSL codebase.

### B. Release strategies

Reconciling software package EOL with OS distribution EOL is a challenging task w.r.t. both feature set and security. For example, consider RedHat Enterprise Linux (RHEL) version 7 with a 10 June 2014 release date. At that time, the stable version of OpenSSL was 1.0.1 and RHEL7 shipped with that package. Once a distribution chooses a software package version, this is essentially written in stone—the stock

TABLE I

Selective OpenSSL features across versions. X25519 only appears since version 1.1.0. Ed25519 support was added in OpenSSL version 1.1.1, released on September 11, 2018.

| | 1.0.1 | 1.0.2 | 1.1.0 | 1.1.1 |
|---|---|---|---|---|
| nistz256 | — | ✓[a] | ✓ | ✓ |
| X25519 | — | — | ✓ | ✓ |
| Ed25519 | — | — | — | ✓ |

[a] Only for x86-64 platforms. On other 64-bit platforms a 64-bit portable C implementation based on [24] is available as a non-default compilation option. Any other platform uses the default generic implementation.

version of OpenSSL on RHEL7 started at 1.0.1 and will stay at 1.0.1 until RHEL7 EOL through 2024. Security issues will be patched with backported fixes and applied to this distribution's OpenSSL flavor by the OS vendor. The reason the version is fixed is that other applications will link against e.g. shared libraries and, since there is no guarantee newer versions will maintain backwards compatibility, the simplest solution is not to change the version number.

Considering bugs, it is an extremely challenging task for the OS vendor to determine which fixes to backport. On the security side, they essentially rely on the risk analysis that takes place during the responsible disclosure process. This process is perilous—e.g. the previously discussed P-256 defect exploited by [14] was not flagged by either the OpenSSL security team or the OS vendors as a security issue, and hence went unpatched over four years in the wild before backported fixes rolled out to address CVE-2011-4354.

This task is slightly easier when the software in question is still maintained and has not reached EOL—the burden falls upon the software package project in question. But what happens when the software package reaches EOL before the OS does? This implies that official fixes are not available from the software package project, and the vendors must rely on contributed and third-party fixes and security analysis.

Before OpenSSL 1.0.2, OpenSSL had no official release strategy and roadmap. Understandably, this made the OS vendor's job difficult, as there was no guarantee on how long official security fixes would be available. With the new release strategy at the end of 2014, OS vendors can make more informed choices regarding OpenSSL version inclusion, yet there will still be a coverage gap. For example, consulting Table I and Figure 1, stock OpenSSL on RHEL7 will not feature X25519 or Ed25519, and since OpenSSL 1.0.1 reached EOL at the end of 2016, RedHat is on the hook for backporting third-party security fixes until the end of 2024 as the OpenSSL team will no longer provide official security fixes.

To summarize, as Figure 1 shows the expected lifetime of OpenSSL versions is in fact much longer than the official EOL stated by OpenSSL. Compounded with the feature sets in Table I, OS vendors for currently deployed distributions are essentially stuck with little to no OpenSSL support for emerging cryptography standards such as X25519 and Ed25519.

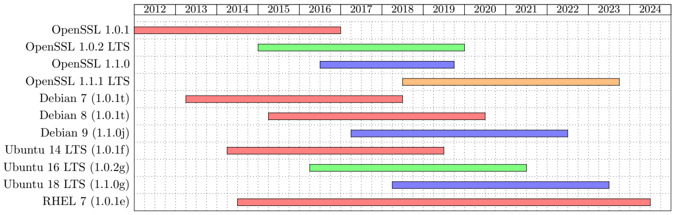

Fig. 1. OpenSSL release strategy vs. OS vendor release strategy: OS version EOL exceeds the OpenSSL version EOL.

## III. OpenSSL and the `ENGINE API`

OpenSSL is an open source project consisting of a general-purpose cryptographic library, an SSL/TLS library and toolkit and a collection of command line tools to generate and handle keys, certificates, PKIs and other cryptographic objects and execute cryptographic operations.

The project officially started in December 1998, with release 0.9.1c forking the SSLeay project by Eric Andrew Young and Tim Hudson. Since then the project has seen a total of thirteen major releases, of which three are currently actively supported: (1) 1.0.2 is the old Long Term Support version, initially released in January 2015 and supported until the end of December 2019; (2) 1.1.0 is the old stable version, released in August 2016 and officially supported in security-fix only mode until September 2019; (3) 1.1.1 is the current stable version and the latest LTS release, realeased on September 11, 2018 after a long development cycle.

Being ubiquitous on the server side (e.g. powering the HTTPS support for the Apache and nginx web servers which combined cover almost two-thirds of Internet active sites according to NETCRAFT's July 2017 Web Server Survey[4]) and in many client tools, the OpenSSL project has become a de facto standard for Internet security.

The project is written mainly in the C programming language and assembly for optimized implementations and supports a plethora of platforms, including a wide range of hardware architectures running most Unix and Unix-like operating systems, OpenVMS and Microsoft Windows.

### A. Architecture of the OpenSSL project

Figure 2 depicts the current architecture of the OpenSSL project. It is based on the current 1.1.1 version, but can be applied with minor changes also to the 1.0.2 and 1.1.0 release branches.

The three main blocks in the diagram are the OpenSSL binaries, consisting of the command-line tools included in the project, which are linked against the other two main blocks of the OpenSSL project diagram, representing the two software libraries implementing the core project functionality.

OpenSSL `libssl` implements the SSL/TLS library, dealing with the implementation details of the standardized network protocols and extensions, linking against OpenSSL

[4] https://news.netcraft.com/archives/2017/07/20/july-2017-web-server-survey.html

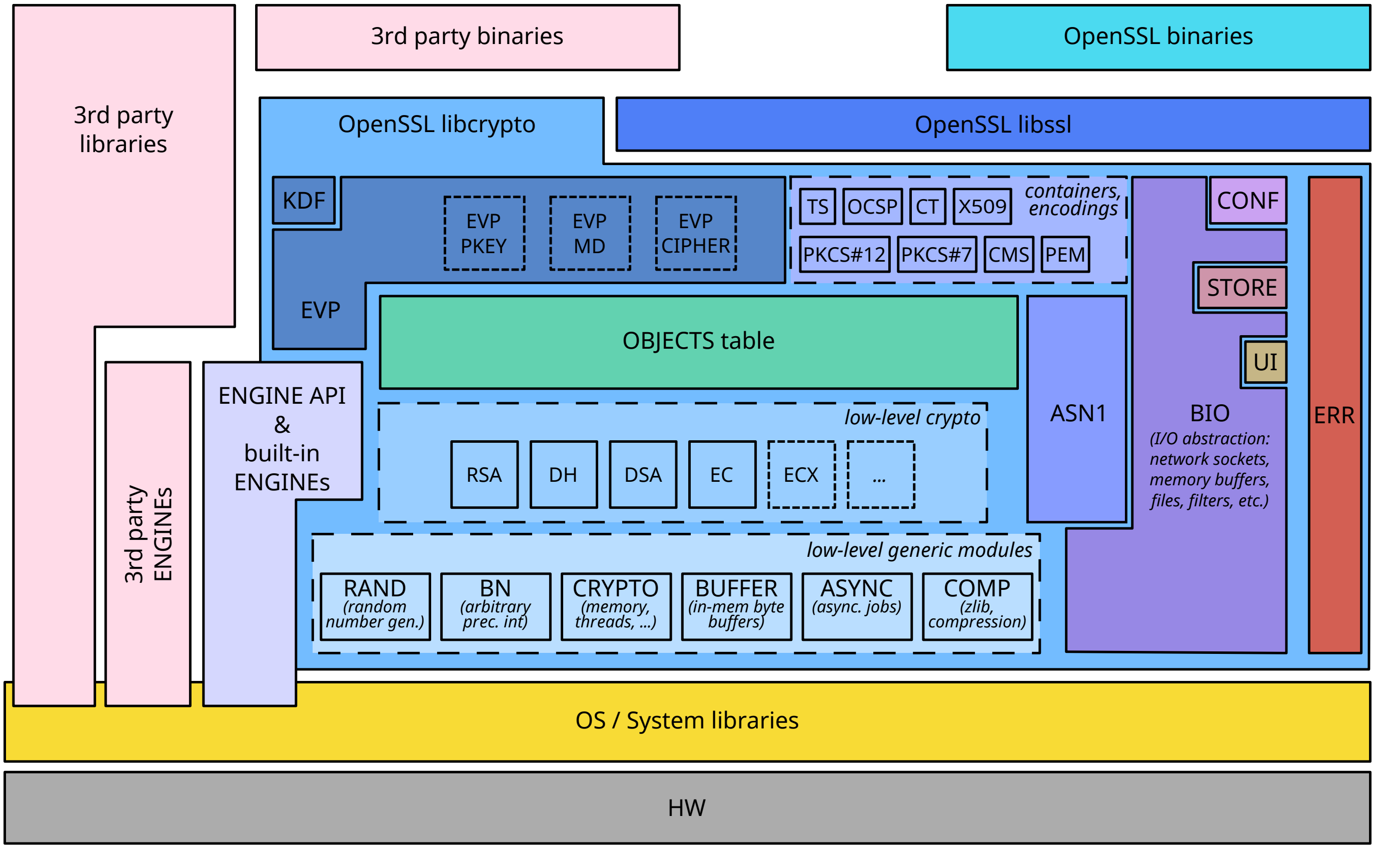

Fig. 2. Architecture diagram of the OpenSSL project.

libcrypto for the implementation of the underlying cryptographic functionality required by the network protocols.

OpenSSL libcrypto contains the actual cryptographic implementations and also acts as portability layer across the different platforms supported by the OpenSSL project, and is further organized into several independent modules.

Focusing on the OpenSSL libcrypto block, the diagram is organized so that, in general, the vertical position is correlated with the level of abstraction of each depicted module (going from the top application-oriented modules to the bottom modules providing lower-level implementations or low-level functionality mostly dealing with operating systems and system libraries abstractions), while the horizontal position, with exceptions, is coupled with the "topic" of each module or group of modules (the left part of the libcrypto block is more closely related to cryptographic implementations, while the right part comprises increasingly general purpose modules).

From a point of view of an application wishing to use OpenSSL to perform cryptographic operations including encryption and decryption, digital signature schemes, key derivation algorithms, and message digests, the high-level EVP API is the recommended interface: the EVP block provides an abstraction level to handle these operations using abstract methods and data types to decouple application programmers from the details of the actual low-level implementations of each cryptosystem. In particular, among the sub-modules included in the EVP API,[5] EVP_PKEY abstracts asymmetric cryptosystems, EVP_MD abstracts message digest cryptosystems, and EVP_CIPHER abstracts symmetric encryption/decryption cryptosystems.

These abstractions are made possible by the use of abstract *context* and *method* structures which describe each EVP cryptosystem in terms of pointers to metadata structures describing its parameters, the functions to manipulate such metadata structures or derived data structures describing the internal status of any of its instances, and the functions operating on such instances to implement the actual cryptosystem functionality.

Therefore, most EVP API functions ultimately act as wrappers around the library internal OBJECTS table, which can be queried by a numeric identifier (NID) to retrieve the actual method structures associated with a particular indexed cryptosystem.

This indirection mechanism can also be used as a way to provide multiple alternative implementations for a given cryptosystem, by manipulating the querying algorithm to select among the method structures registered for the same NID: this approach is what ultimately powers the ENGINE API described in the following paragraphs.

[5] https://www.openssl.org/docs/man1.1.1/crypto/evp.html

### B. What is an OpenSSL ENGINE?

OpenSSL 0.9.6 introduced a new component to support alternative cryptography implementations, most commonly for interfacing with external crypto devices (e.g. accelerator cards), in the form of ENGINE objects.[6]

These objects act as “containers for implementations of cryptographic algorithms” and can be statically linked in the OpenSSL library at compile-time or dynamically loaded at run-time in and out of the running application with low-overhead from external binary objects implementing the ENGINE API[7] through a special built-in ENGINE called “dynamic”.

Such dynamic ENGINEs are particularly interesting due to the flexibility they provide: (1) they allow to replace compiled-in implementations affected by known problems with newer ones, maintaining compatibility with existing applications; (2) they allow hardware vendors to release self-contained shared-libraries to add support for arbitrary hardware to work with applications based on OpenSSL, keeping their software outside of the main OpenSSL codebase; (3) they allow to reduce the memory impact of OpenSSL, by avoiding to statically link support for unneeded hardware at compile-time in favor of system configuration or automatically probing for supported devices at run-time and dynamically loading only the required cryptographic modules.

Dynamic ENGINEs are also interesting for software implementations: (1) they allow to replace compiled-in software implementations in case of bugs, vulnerabilities or sub-optimal performances with newer alternative implementations; (2) they provide an alternative in case of issues with the OpenSSL core development team decision-making process, decoupling the decision to adopt the OpenSSL library from the choice of individual cryptosystem implementations, without incurring the costs of maintaining a fork of the OpenSSL project or patch sets, while also providing transparent binary compatibility with existing applications; (3) they allow to backport newer cryptosystems in previous versions of the OpenSSL library and existing applications based on it; (4) they allow to easily add new cryptosystems or new implementations for already compiled-in cryptosystems to the OpenSSL library, providing a convenient way to test and benchmark new software implementations in a real-world context; (5) they offer a greater degree of freedom from the OpenSSL project toolchain, allowing developers to use different programming languages and build systems, further lowering the development and maintenance costs for developing plugin alternative implementations or new functionality; (6) they offer flexibility to solve licensing issues: currently the OpenSSL project is released under a "dual license" scheme, under the OpenSSL License, (a derivate of the Apache License 1.0) and the SSLeay License (similar to a 4-clause BSD License), and is in the process of transitioning to the Apache License 2.0. Contributors are thus forced to release their work under these licenses, which may be an issue especially when reusing code from projects released under a proprietary license or an incompatible copyleft license. Being objects dynamically loaded at runtime, engines can benefit from usually more flexible licensing requirements, providing a bridge towards software released under different licenses.

*Functionality provided by ENGINEs:* The cryptographic functionality that can be provided by an ENGINE implementation includes the following abstractions: (1) RSA_METHOD, DSA_METHOD, DH_METHOD, EC_METHOD; providing alternative RSA/DSA/etc. implementations; (2) RAND_METHOD: providing alternative (pseudo-)random number generation implementations; (3) EVP_CIPHER: providing alternative (symmetric) cipher algorithms; (4) EVP_MD: providing alternative message digest algorithms; (5) EVP_PKEY: providing alternative public-key algorithms.

### C. Anatomy of a dynamic ENGINE

At the highest level of abstraction, a dynamic ENGINE can be split into two functional blocks.

One block contains all the alternative implementations for the cryptosystems provided by the ENGINE: this part mainly consists of a collection of structs for each cryptosystem, each linking to the actual functions implementing its operations. For example, an EVP_MD message digest struct would reference the actual init(), update(), and final() functions implementing the OpenSSL message digest streaming API, in addition to some utility functions allowing the OpenSSL library to cleanly handle, clone and destroy instances of the provided message digest implementation.

Every such struct would be individually registered against the OpenSSL library during the bind() process, and structs of the same kind (e.g. all the EVP_MD structs, all the EVP_PKEY_meth structs, etc.) are glued together by functions registered in the ENGINE object that allow the OpenSSL library to query the engine for lists of provided algorithms or a specific algorithm indexed by NID.

The other block contains the bind() method and the initialization and deinitialization functions: (1) the bind() method is called by the OpenSSL built-in dynamic ENGINE upon load and is used to set the internal state of the ENGINE object and allocate needed resources, to set its id and name, and the pointers to the init(), finish(), and destroy() functions; (2) the init() function is called to derive a fully initialized functional reference to the ENGINE from a structural reference; (3) the finish() function is called when releasing an ENGINE functional reference, to free up any resource allocated to it; (4) the destroy() function is called upon unloading the ENGINE, when the last structural reference to it is released, to cleanly free any resource allocated upon loading it into memory.

### D. Curve25519, X25519 and Ed25519 in OpenSSL

OpenSSL 1.1.0 introduced support for X25519: as a consequence of the way Curve25519 and X25519 are defined, instead of adding the curve inside the EC module containing

[6] https://github.com/openssl/openssl/blob/OpenSSL_1_1_1-stable/README.ENGINE

[7] https://www.openssl.org/docs/man1.1.1/man3/ENGINE_init.html

every other elliptic-curve cryptosystem implementation based on prime or binary fields, the OpenSSL developers decided to add a dedicated ECX sub-module defining EVP_PKEY_-meth structures directly linking to a self-contained portable C implementation of Curve25519 and the underlying finite field arithmetic.

The actual low-level portable C implementation was closely based on the ref10 version of Ed25519 in SUPERCOP 20141124 which, although portable, suffers huge speed penalties compared to implementations optimized for specific architectures or even portable 64-bit C code.

The latest release of OpenSSL (1.1.1) expanded the ECX sub-module to add a new EVP_PKEY_meth supporting the Ed25519 digital signature scheme (based on a seperate portable C implementation). During its development cycle, the X25519 low-level scalar multiplication was also revised, adding a specialized portable 64-bit C implementation, and an assembly implementation for x86_64.

As a result, comparing the benchmarks of X25519 (in OpenSSL 1.1.0) and Ed25519 (in OpenSSL 1.1.1) cryptosystems with operations of analogous EC cryptosystem (e.g. over the NIST P-256 elliptic curve) does not yield the performance benefits expected from [4, 5], as shown later in Section V.

## IV. The libsuola ENGINE

libsuola[8] takes advantage of the ENGINE API described in Section III to provide alternative software implementations for X25519 and Ed25519, working as a bridge between OpenSSL and an external library.

libsuola itself is a shallow loadable module, i.e. it does not contain cryptographic implementations: the core part of libsuola is the code facing the OpenSSL APIs, that takes care of initializing data structures and registering abstract methods; these abstractions are routed to a "provider" module in libsuola, which finally links against the selected external library to provide the actual cryptographic functionality. To demonstrate the potential of the proposed methodology, we provide three different providers, among which the user can select at build time, each linking against a different implementation, and further described in Section IV-D.

Figure 3 shows the architecture of libsuola (compiled selecting the libsodium provider) and its interactions with libsodium and OpenSSL. Selecting a different provider results in linking against another library or embedding, through static linking, an implementation inside the provider object, but the changes affect only the provider object, which is the only element providing cryptographic functionality.

For the installed applications that will benefit from the added functionality or alternative implementations, libsuola is completely transparent: at run time when OpenSSL is loaded by the application, the library reads from system-wide configuration files, and is instructed to subsequently load the libsuola ENGINE. Alternatively, through environment variables, it is possible to have application-specific OpenSSL configuration files or programmatically use the OpenSSL API from application code to explicitly load libsuola.

Independently of the way it is instructed to load the libsuola ENGINE, internally the OpenSSL library creates an instance of the built-in *dynamic* ENGINE, which in turn uses the OS dynamic loader to load libsuola.

As soon as it is loaded into memory, the *dynamic* ENGINE calls the libsuola suola_bind() function which in turn: (1) sets the id and name fields in the ENGINE structure; (2) sets pointers to the destroy(), init() and finish() functions in the ENGINE structure; (3) registers NIDs for X25519, Ed25519 and the *identity message digest*: NID_-X25519 is defined in OpenSSL since version 1.1.0, while NID_ED25519 is only defined in 1.1.1. This step ensures that even in previous versions of OpenSSL the internal OBJECTS table includes definitions for both cryptosystems; (4) creates the structures to handle each implemented cryptosystem: the EVP_MD md_identity describing the *identity message digest* and two (EVP_PKEY_meth, EVP_PKEY_ASN1_meth) pairs of structures for X25519 and Ed25519. Each of these structures is also registered in the internal OpenSSL OBJECTS table under the corresponding NID; (5) sets pointers to the digests(), pkey_meths() and pkey_asn1_meths() callbacks in the ENGINE structure: these callbacks allow the OpenSSL library to manipulate the ENGINE handle querying for lists of implemented methods indexed by NID; (6) calls the provider suola_implementation_init() function to initialize the internals of the provider implementation.

The rest of libsuola consists of the collection of functions used to implement the functionality described by each of the structures registered in the suola_bind() function, which map OpenSSL structures to the provider functionality. These are briefly described in the following paragraphs.

### *A. libsuola X25519*

X25519 is implemented through an EVP_PKEY_meth and a corresponding EVP_PKEY_ASN1_meth. The first one mainly includes the definition of a keygen() and a derive() method, while the second includes methods for encoding and decoding X25519 private and public values.

The keygen() method delegates its core functionality to the suola_scalarmult_curve25519_base() function, which is then routed to the actual implementation through the selected provider.

The derive() method, called once the peer key has been set, performs a generic point scalar multiplication over Curve25519 through the provider suola_scalarmult_-curve25519() function.

### *B. libsuola Ed25519*

Similarly, Ed25519 is also implemented through an EVP_-PKEY_meth and a corresponding EVP_PKEY_ASN1_meth. The first one mainly includes the definition of a keygen(), a sign() and a derive() method; the second includes methods for encoding and decoding Ed25519 private and public values, as well as a *control* method that the OpenSSL

[8] https://github.com/romen/libsuola

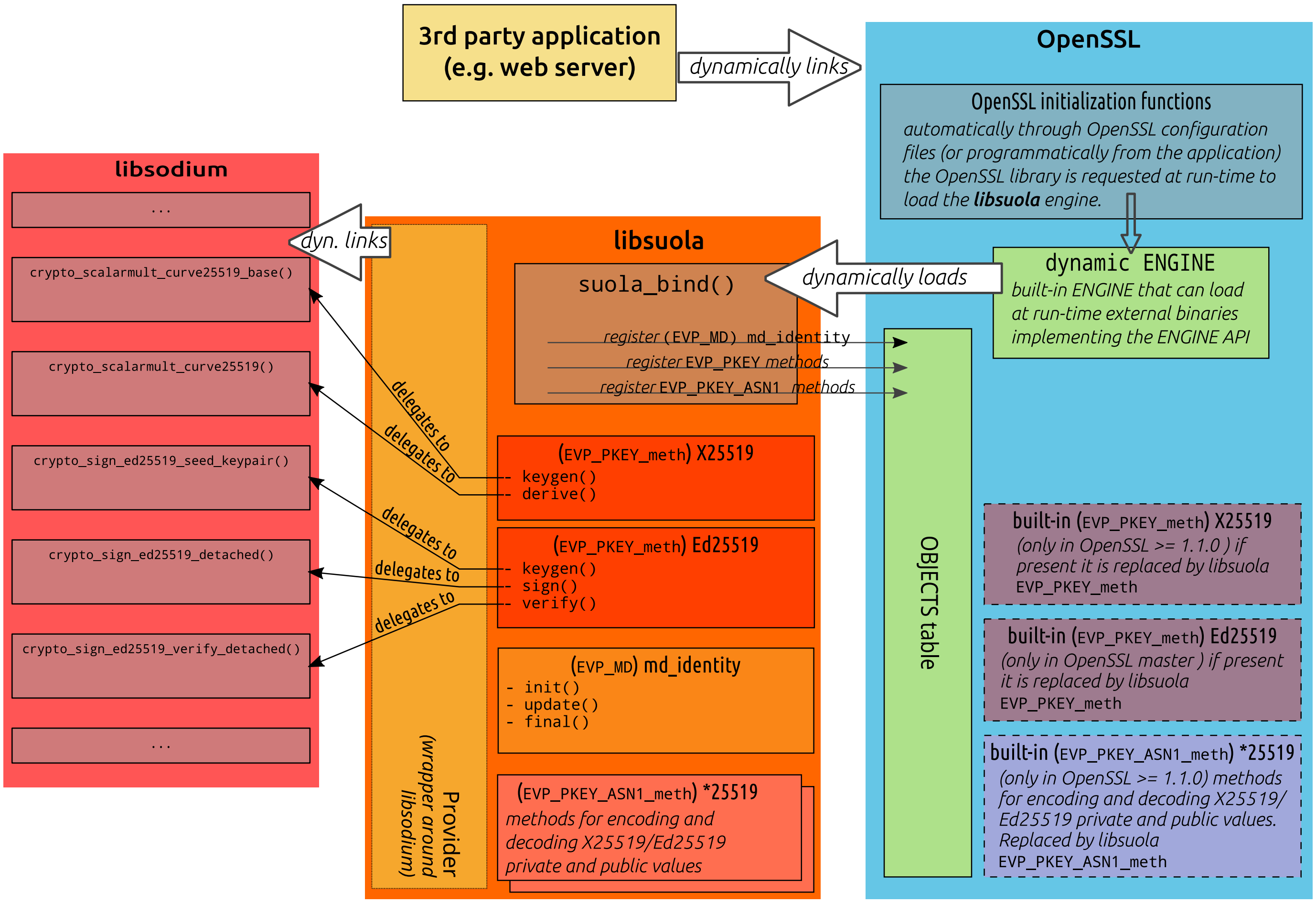
3rd party application (e.g. web server)
dynamically links
OpenSSL
OpenSSL initialization functions
automatically through OpenSSL configuration files (or programmatically from the application) the OpenSSL library is requested at run-time to load the libsuola engine.
libsodium
...
crypto_scalarmult_curve25519_base()
crypto_scalarmult_curve25519()
crypto_sign_ed25519_seed_keypair()
crypto_sign_ed25519_detached()
crypto_sign_ed25519_verify_detached()
...
dyn. links
libsuola
suola_bind()
dynamically loads
dynamic ENGINE
built-in ENGINE that can load at run-time external binaries implementing the ENGINE API
register (EVP_MD) md_identity
register EVP_PKEY methods
register EVP_PKEY_ASN1 methods
OBJECTS table
delegates to
delegates to
delegates to
delegates to
delegates to
(EVP_PKEY_meth) X25519
- keygen()
- derive()
(EVP_PKEY_meth) Ed25519
- keygen()
- sign()
- verify()
(EVP_MD) md_identity
- init()
- update()
- final()
(EVP_PKEY_ASN1_meth) *25519
methods for encoding and decoding X25519/Ed25519 private and public values
Provider (wrapper around libsodium)
built-in (EVP_PKEY_meth) X25519
(only in OpenSSL >= 1.1.0 ) if present it is replaced by libsuola EVP_PKEY_meth
built-in (EVP_PKEY_meth) Ed25519
(only in OpenSSL master ) if present it is replaced by libsuola EVP_PKEY_meth
built-in (EVP_PKEY_ASN1_meth) *25519
(only in OpenSSL >= 1.1.0) methods for encoding and decoding X25519/ Ed25519 private and public values. Replaced by libsuola EVP_PKEY_ASN1_meth

Fig. 3. `libsuola` architecture and its interactions with `libsodium` and OpenSSL.

library uses to query for the default message digest algorithm associated with the digital signature cryptosystem, which in `libsuola` is set to return the `NID` for `md_identity`.

The `keygen()` method for Ed25519 is mostly a wrapper around the provider `suola_sign_ed25519_seed_keypair()` function.

The `sign()` and `verify()` methods are wrappers around `suola_sign_ed25519_detached()` and `suola_sign_ed25519_verify_detached()` functions in the provider. These functions implement a *PureEdDSA* signature scheme, i.e. where the message to be signed is directly used as an input to the signature generation algorithm, without a pre-hashing step as opposed to traditional digital signature schemes based on RSA, DSA or ECDSA or to the *PreHashEdDSA* signature scheme.

OpenSSL 1.1.1 introduced new API functions to support one-shot signature generation and verification cryptosystems like Ed25519, but using this approach in `libsuola` would make it incompatible with previous versions of OpenSSL. Instead, we chose to work around the limitations of the traditional API by adding a custom `md_identity` message digest method to be used in the default pre-hash step performed by OpenSSL before the actual signature generation or verification.

### C. `libsuola md_identity`

The `EVP_MD md_identity` is a custom message digest algorithm behaving as an identity function to be used as the pre-hash algorithm in the `libsuola` implementation of PureEd25519: the output of this message digest is a copy of the input message.

The `EVP_MD` API in OpenSSL can be reduced to a *streaming* approach based on calling an `init()` function before starting the hash computation, repeated calls to an `update()` function until the whole input message is consumed, and a final call to a `final()` function to finalize the hash and retrieve its output. For `md_identity` these functions are implemented as described below:

**`init()`** Securely allocates an empty small buffer as the internal status of the message digest algorithm, tracks its length and sets to 0 the counter of used buffer bytes;

**`update()`** Depending on the size of the input block and the availability of unused space, the internal buffer is securely reallocated to sufficiently increase its size. A copy of the new input block is then concatenated to the existing data in the buffer while tracking the length of the whole buffer and of the data contained in it;

**final()** OpenSSL EVP_MD API limits the maximum size of a message digest algorithm output thus, considering that no actual finalization is required and that the md_identity algorithm is used only by code inside libsuola, we decided to work around this limit by gaining access to the internal md_identity buffer directly from the Ed25519 EVP_PKEY_meth implementation, without ever using the final() function. As a result, the implementation of this function simply returns an error.

### D. Providers

The actual cryptographic functionality provided by libsuola is entirely contained in the provider module. It determines which library libsuola is linked against, and routes the function calls from the OpenSSL structures described above to the relevant functions in the selected implementation.

To demonstrate the potential of the proposed methodology, we provide three different alternative providers: (1) the first provider, to which Figure 3 refers, links against libsodium, a fork of NaCl; (2) a second provider links against the HACL* library, providing a formally verified implementation, through an API compatible with NaCl; (3) as a proof of concept, we develop a third provider which does not depend on an external library and internally embeds the provided functionality through static linking.

*libsuola-sodium:* For our first provider, libsodium was selected as a modern cryptographic library, designed emphasizing high security and ease-of-use, and addressing from its inception as a fork of NaCl many of the shortcomings we described in OpenSSL.

While the aim of our project is to provide support for emerging cryptography standards such as X25519 and Ed25519 for currently deployed versions of OpenSSL, as detailed in Section V, we also notice that, on the platforms we tested, the alternative implementations provided through libsodium generally performs equally or better than the built-in implementations included in the previous OpenSSL 1.1.0 release (supporting X25519 only). Even in the latest 1.1.1 release of OpenSSL (supporting both X25519 and Ed25519), the libsodium Ed25519 implementation provided through libsuola appears to be faster than the OpenSSL default implementation.

When this provider is selected, as depicted in Figure 3, libsuola is dynamically linked against libsodium, hence when OpenSSL loads the ENGINE, the dynamic loader would automatically load both libsuola and libsodium.

The suola_implementation_init() function in this case calls sodium_init() during the binding process to initialize libsodium internals.

*libsuola-hacl:* HACL* [2] is a very interesting target for our framework, as it presents a practical implementation of a cryptographic library, which is formally verified for memory safety and functional correctness with respect to its published standard specification, and aims to be "as fast as state-of-the-art C implementations, while implementing standard countermeasures to timing side-channel attacks".

Through our proposed methodology it then becomes possible to integrate the guarantees of projects like this, that mathematically prove the absence of the defects we listed in Section II-A, without needing to alter existing application to use a different cryptographic library.

HACL* uses the same C API as libsodium for the NaCl constructions, so excluding some obvious prefix renaming, the mappings described in Figure 3 with respect to libsodium are analogous to the ones provided by this provider for the HACL* library.

*libsuola-donna:* As a proof of concept, we implemented an alternative version of libsuola that statically links at compile-time against an implementation of Curve25519 and X25519 based on the work of Adam Langley and Andrew Moon.[9]

Through this provider, the contents of the EVP_PKEY_meth structures for Ed25519 and X25519 are routed as follows: (1) the keygen() functions ultimately link to, respectively, ed25519_publickey() and curved25519_scalarmult_basepoint() from floodyberry/ed25519-donna/ed25519.c; (2) the Ed25519 sign() and verify() functions are implemented via, respectively, ed25519_sign() and ed25519_sign_open() from floodyberry/ed25519-donna/ed25519.c; (3) the X25519 derive() function is not supported by the code in the floodyberry/ed25519-donna repository, so we opted to use the portable 64-bit C *donna* implementation included in SUPERCOP[10] in crypto_scalarmult/curve25519/donna_c64/smult.c. This implementation is not compatible with our 32-bit ARM testing environment, so on this platform at build time we replace it with another implementation from SUPERCOP, in crypto_scalarmult/curve25519/neon2/scalarmult.s, contributed by Bernstein and Schwabe [32], optimized for the NEON Advanced SIMD extension.

## V. Experimental results

Our last contribution consists in an experimental evaluation of the libsuola ENGINE, carried out by benchmarking the provided operations across different versions of OpenSSL on different target architectures, by analyzing how the proposed methodology effectively addresses the concerns listed in Section II and its limits.

### A. Benchmarks

For the benchmarks, for which the detailed results are reported in Appendix A, we targeted two platforms to address both the desktop/server scenario and a mobile/embedded system scenario: a 4-core/8-threads 3.4GHz Intel Core i5-6700 Skylake CPU (Table III) and a quad-core 1.2GHz Broadcom BCM2837 CPU on a Raspberry Pi 3B on a 32-bit (Table IV) and a 64-bit environment (Table V).

[9] https://github.com/floodyberry/ed25519-donna

[10] http://bench.cr.yp.to/supercop.html

We collected the measurements using a benchmarking application derived from the OpenSSL `speed` app, modified to use the `EVP API` for every operation and to measure CPU cycles for a fixed number of repeated runs of the specified operations (in contrast with measuring the number of repeated runs over a fixed amount of time). We linked the benchmarking application, and `libsuola` in its three different versions (i.e. one for each provider selected at compile time), against OpenSSL versions 1.0.2r (old LTS), 1.1.0j (old stable) and 1.1.1b (current LTS release).

The table reports the execution cost in CPU cycles for the following cryptosystem operations: key generation, ECDH shared secret derivation, digital signature generation, and digital signature verification. As a baseline reference, the table also presents the execution cost of these operations for cryptosystems based on the fast `nistz256` implementation for the popular NIST P-256 elliptic curve.

The benchmarks demonstrate that our methodology achieves the goal of adding missing functionality to OpenSSL transparently for existing applications, and to replace the default implementations with alternative ones.

Excluding X25519 primitives in OpenSSL 1.1.1b, we also discovered that using `libsuola-sodium`, on top of adding missing functionality, generally also improves the performance of the listed primitives, as `libsodium` selects implementations that are more optimized for the test platforms.

We also note that, as claimed in [2], HACL* does generally achieve relatively good performance, comparable with the default implementations in OpenSSL, and notably in the case of X25519 `derive()` in OpenSSL 1.1.0j, the HACL* implementation is even twice as fast as the default implementation on x86-64.

### B. Analysis and security evaluation

In Section I and Section II we listed a series of concerns regarding software assurance, release strategies and limits of OpenSSL as a platform for researchers to motivate our work. In this section we analyze how our proposed methodology addresses those concerns and its limits.

*Software assurance:* We claim that the `ENGINE` approach is useful in preventing various vulnerabilities that plague OpenSSL, including e.g. traditional software defects, arithmetic defects in e.g. hand-coded assembly, and various side-channel attack vectors. To support this claim, we use the following metric. We first enumerate all the CVEs issued by OpenSSL and then fetch the related security metadata for each CVE. To semi-automate this, we utilize the Computer Incident Response Center Luxembourg (CIRCL) database[11] that provides a contextual feed of security vulnerabilities. We then count the number of security advisories issued across selected vendors. We partially validated the results by examining the Debian and Ubuntu patches applied to package builds.

As a case study, we did the same analysis for `libsodium`, for which we found no CVEs issued—which does not mean there are no vulnerabilities, but is consistent with no patches found in said package builds. We carried out our analysis separately for all OpenSSL versions which are currently not EOL: 1.0.2, 1.1.0, and 1.1.1. Summarizing the Table II results, we conclude that our approach of dedicated backend crypto providers for an `ENGINE` do not suffer from the same classes of security issues as the analogous functionality in `libcrypto`, supporting our claim.

TABLE II
OpenSSL security statistics across OS vendors, as of 08 April 2019. The advisory counts are restricted to CVEs affecting `libcrypto`.

| OpenSSL version | 1.0.2 | 1.1.0 | 1.1.1 |
|---|---|---|---|
| CVEs, total | 63 | 23 | 3 |
| CVEs, `libssl` | 22 | 9 | 0 |
| CVEs, `libcrypto` | 41 | 14 | 3 |
| Amazon Linux Security Advisories | 16 | 8 | – |
| Amazon Linux 2 Security Advisories | 3 | 3 | 1 |
| Debian Security Advisories | 14 | 8 | 2 |
| Debian LTS Advisories | 9 | 5 | 1 |
| Fedora Security Updates | 42 | 18 | 1 |
| FreeBSD Security Advisories | 29 | 14 | 2 |
| Gentoo Linux Security Advisories | 11 | 5 | – |
| Oracle Security Advisories | 22 | 7 | – |
| Redhat Security Advisories | 33 | 12 | – |
| Slackware Security Advisories | 15 | 6 | 1 |
| SUSE Security Updates | 81 | 47 | 10 |
| Ubuntu Security Notices | 15 | 7 | 1 |

It can also be argued that `libsuola` itself is adding even more surface for bugs and defects, that might result in additional risks. While this is definitely true for any additional line of code in a software system, we designed `libsuola` as a "shallow" `ENGINE` to minimize the risk of introducing such defects: `libsuola` itself is not providing any cryptographic functionality, and we designed it to maximize readability and maintainability making it modular.

Also, compared with the process of patching OpenSSL directly to add missing or alternative primitives, modifying our proposed `ENGINE` template is generally an easier process: the actual cryptographic functionality can be implemented using any language and build system as long as it produces C bindings that can be mapped in a dedicated "provider" module, while the `ENGINE` functionality remains separate and mostly reusable, which further minimizes the risks of introducing software defects.

*Release strategies:* One of our original goals was to facilitate the integration of missing functionality in end-of-life, yet still widely deployed, versions of OpenSSL, motivated by different release strategies applied by software providers in real-world systems as described in Section II-B.

We demonstrated how the proposed approach allows to easily add or backport functionality in OpenSSL 1.0.2, and how this is done transparently for existing applications.

Alternatives to our approach usually require to manually recompile and install more recent versions of OpenSSL and then repeat the same process for each component of the software stack of the target system that depends on OpenSSL. Sometime this is not even possible if existing applications do

[11] https://www.circl.lu/services/data-feeds-cve/

not support the newer release of OpenSSL, in which case the system administrator would need to design a patch (or retrieve a trusted one) for the target software. This approach is impractical in most medium- or large-scale deployments, as it is costly to maintain and definitely adds even more surface for the rise of critical software defects and security vulnerabilities.

The same can be told about planning to replace the cryptographic provider for existing applications based on OpenSSL. Excluding some exceptions where software is designed with a "cryptographic module abstraction layer" to easily swap the default cryptographic module with alternative supported ones, most applications do not usually allow this level of flexibility, and extensive patches would be required to modify, for example, an application using OpenSSL to serve TLS connections to use a different library for the TLS stack supporting `libsodium` as the cryptographic primitive provider.

*Accessibility for researchers:* Another goal of our research listed in Section I aimed at delivering a framework to enable researchers to do testing and benchmarks in real-world scenarios, and to easily disseminate novel implementations and primitives to a larger user audience.

Our proposed methodology achieve this by lowering the development and maintenance costs of adding functionality to the widely used OpenSSL library.

Freedom of choice in programming languages and building tools for the actual cryptographic implementation makes it easier to plug functionality into OpenSSL. It also lowers the long term maintenance costs, especially considering the usually long and possibly unfruitful process of proposing the addition of novel or alternative functionality into the main OpenSSL codebase. Moreover, the higher degree of freedom about licensing allows to further reduce the entry barrier for adding functionality to applications based on OpenSSL.

Finally, it provides an interesting framework to collect real data about researchers' novel contributions, which are readily comparable with existing functionality by reusing existing benchmarking facilities. Moreover, being an optional component and easy to plug in at run-time, it makes it easier for researchers to reach a wider user audience for more extensive testing or for releasing their projects.

### C. Limits

*The scope is limited to OpenSSL:* First and foremost, our proposed methodology applies to OpenSSL only. While it can be argued that this is sufficient considering the portion of the whole Internet that currently depends on OpenSSL as part of its security stack, we recognize that this is a strong limit of our proposed methodology. In Section VI we discuss potential research directions to overcome this limit in the future.

*Overhead of linking to other libraries:* Another limit of our approach is the memory consumption: when using `libsuola` to integrate missing functionality in OpenSSL, in addition to the memory required by OpenSSL itself, `libsuola` and the library it depends on (`libsodium` or HACL*) are also loaded into memory.

One can argue that this cost should be justified, e.g. in comparison with loading into memory only the library actually providing the cryptographic implementation (in our case either `libsodium` or HACL*). It is true that if the application was rewritten to only use the primitives in `libsodium` —i.e. directly linking against `libsodium` —the memory cost would be more than halved compared to our approach. However: (1) this would require redesigning every single application that currently depends on OpenSSL, defying our goal of providing the additional functionality transparently to existing applications; (2) part of the design of `libsodium` (and NaCl) is based on providing only an opinionated set of primitives, therefore applications patched to rely exclusively on `libsodium` for their cryptographic needs would lose generality and compatibility with other applications, which is often not an option; (3) `libsodium` only provides cryptographic primitives, so redesigning the applications as has been proposed would also incur the additional cost (and security risks) of selecting and adapting to yet another library to implement the protocol functionality (e.g. TLS) on top of `libsodium`.

For these reasons we do not believe that such comparison is fair, because what forks of NaCl gain in elegance, conciseness, ease-of-use, performance and security comes at the cost of not being interoperable with projects that do not implement the same set of primitives, and with both the benefits and the drawbacks of being only a cryptographic library and not a full stack library.

*`libsuola` memory overhead:* Regarding the overhead of `libsuola` itself, with respect to memory consumption, its footprint is negligible compared with the memory requirements of OpenSSL across all the tested architectures and OpenSSL versions.

To give some figures, in the case of our x86_64 target platform and OpenSSL 1.1.1, the resident set size in memory of `libsuola.so` alone totals 60 KB, compared with over 2 MB of memory occupied by OpenSSL alone (excluding the additional 1.2 MB occupied by `libc`, `libpthread`, and `libdl` which are required by OpenSSL).

*`libsuola` computational overhead:* With respect to computation, excluding from our analysis `md_identity` which is not of particular relevance, `libsuola` itself adds some computational overhead once when it is dynamically loaded by OpenSSL during initialization, and a small computational cost every time the functions wrapping the actual cryptographic primitives in the provider module are called.

OpenSSL uses exactly the same level of indirection whether the function implementing a primitive resides in OpenSSL itself or in an external `ENGINE`. The additional cost comes from the fact that the implementing function is resolved with the memory address of the function in the provider object inside `libsuola`, which then calls the actual function in `libsodium`, HACL* or the internal object with the `donna` implementation.

With the `libsuola-donna` case the wrapper is actually required as the `donna` implementation does not implement the NaCl C API, but in the case of libsuola-sodium and libsuola-hacl, where the wrapper only calls the relevant function in the target library, this overhead could be erased by annotating the source code of the providers with compiler attributes to instead resolve the final function pointer at load time, so that OpenSSL would call directly the relevant function in the target library.

This further optimization would come at the cost of higher complexity, inferior code readability and limited portability, which we deemed to be not in line with the stated goals of our project. We also believe that researchers who strongly need to squeeze out every single superfluous assembly instruction, while testing their code against OpenSSL through our proposed methodology, will have no problem implementing the described expedient.

## VI. Related work

We focused our analysis on the OpenSSL project and on ENGINEs as the native mechanism to handle alternative cryptographic implementations for the supported primitives. In this section, we present an overview of how other security standards, libraries, and frameworks address extensibility.

Describing programming interfaces and application protocols to provide interoperability and transparency between applications and cryptographic providers in the form of hardware and software cryptographic modules is a decades-old problem. Standardization bodies, operating systems, security software and hardware vendors, manufacturers and various organizations over the years approach the issue aiming for akin goals but with different approaches (e.g. regarding the cryptographic awareness required to adopt the proposed solution), trust models and features.

*Standards:* One of the more relevant and widespread cryptographic standards is PKCS #11 [33]: it specifies the "Cryptoki" API for devices that hold cryptographic information and perform cryptographic functions, presenting applications an abstract "cryptographic token" view, thus providing a separation layer between applications and specific algorithms and implementations. The PKCS #11 API defines abstract object types to represent symmetric and asymmetric crypto keys, digital signature keys, X.509 certificates, hash functions, MACs, and other common cryptographic objects, and all the functions needed to generate, modify, use and delete such objects. Just like OpenSSL ENGINEs, PKCS #11 was originally designed for hardware devices, but the level of abstraction provided allows it to be used also for software cryptographic modules. By design, PKCS #11 defines use cases with many applications and many tokens, allowing interaction and the possibility to choose among alternative implementations.

Other related standards that aim to separate applications from cryptographic implementation details include GSS-API [34, 35] and IDUP-GSS-API [36], CDSA [37], SS-API [38], and the Simple Cryptographic Program Interface [39].

These standards were excluded as an alternative to the presented libsuola approach because currently they are not natively supported by OpenSSL: e.g. PKCS #11 support relies on an external dynamic ENGINE.

*Operating System assisted crypto:* Operating Systems are by design natural candidates to provide security services and abstraction of hardware capabilities to users and applications of a system. Modern OS designs provide a hardware abstraction layer (HAL), separation of security contexts and levels, access control, authentication of loadable modules and binaries, have privileged access to cryptographic units and co-processors, and usually already require a set of cryptographic primitives for their internal functions.

The OpenBSD/FreeBSD Cryptographic Framework (OCF) provides convenient access to the kernel cryptographic functionalities (including symmetric and public-key crypto, hashes, MACs, access to crypto hardware accelerators and RNGs) to userspace through a /dev/crypto pseudo-device [40, 41] and a standard ioctl interface, simplifying the development of applications and reducing the required cryptographic awareness of application developers.

Recent versions of the Linux kernel include the AF_ALG interface (providing access to the kernel symmetric crypto functions through the socket interface), while independent developers maintain a set of patches[12] to provide a /dev/crypto ioctl interface compatible with the OCF one. See [42] for a performance analysis of this framework in different usage scenarios and [43] for a proposal to enhance this framework to provide decoupling of cryptographic keys from applications for increased forward secrecy properties. A different project provides a complete port of OCF to the Linux kernel[13], including the ioctl interface but using the ported implementations instead of the native Linux crypto capabilities.

Microsoft Windows OSs expose a Cryptographic Application Programming Interface (CAPI, a.k.a. CryptoAPI), providing an abstraction layer and a set of dynamically linked libraries decoupling applications from the functionality provided by Cryptographic Service Providers (CSP). The supported functionality includes RNGs, symmetric and public-key crypto, hashes and MACs, authentication, PKI and key storage. The "Cryptographic API: Next Generation" (CNG)[14] is the latest long-term replacement of the original API, providing a compatibility layer and featuring better support for configuration, cryptographic agility, and a wider selection of supported primitives, covering the whole NSA Suite B [44] (including ECC).

Support of OS-provided cryptographic functionality in OpenSSL is generally implemented through dedicated ENGINEs.

*Other security libraries and frameworks:* Mozilla NSS[15] is another widespread set of libraries supporting cross-platform development of security-enabled client and server applications.

[12] http://cryptodev-linux.org

[13] http://ocf-linux.sourceforge.net/

[14] https://msdn.microsoft.com/en-us/library/windows/desktop/aa375276.aspx

[15] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS

It has native PKCS #11 support for hardware and software security modules, and indeed the internal cryptosystem implementations are encapsulated in a software PKCS #11 token, allowing selection among alternative implementations of crypto primitives during configuration.

GnuTLS[16] is a high-level library providing TLS support, and relies on other libraries for crypto primitives. It features a multilayered architecture, based on a Cryptography Provider Layer, which provides abstraction from the actual providers for individual primitives and allows transparent access to implementations provided by the underlying cryptographic software library (i.e. libgcrypt or nettle), to OS assisted crypto (e.g. /dev/crypto or CAPI), non-privileged CPU crypto instructions (e.g. Intel AES-NI), and TPM or hardware and software cryptographic modules through a rich native support for PKCS #11. The architecture also provides functions to override symmetric crypto implementations and “abstract private keys” as a way to handle abstractions over keys stored in hardware modules (PKCS #11 or TPM) or over operations implemented directly using an external API.

ARM mbed TLS[17] (formerly known as PolarSSL) has optional support for Hardware Security Modules (HSM) with PKCS #11, and supports the replacement of specific functions or full cryptosystem modules with alternative implementations, but limited to compile time.[18]

cryptlib [45, 46] has built-in native support for selected hardware crypto accelerators and for PKCS #11 devices, ensuring that the API for any crypto device is identical to the API of cryptlib native crypto implementations, allowing easy and transparent migration of applications from the native software implementations to the use of crypto devices.

The Java SDK includes the Java Secure Socket Extension (JSSE),[19] and the Java Cryptography Architecture (JCA).[20] These modular frameworks provide an abstraction layer respectively for secure network protocols like TLS and for cryptographic primitive implementations. Through the factory method design pattern and the Service Provider Interface (SPI) programming interface, JCA supports the registration of Cryptographic Service Provider (CSP) instances. Applications using the framework can either select a specific CSP for a primitive or let the system configuration select the most suitable implementation at runtime, attaining complete decoupling between applications and crypto implementations.

Other libraries have a completely opposite design philosophy: NaCl and libsodium, for example, are intentionally designed as simplified and opinionated libraries to avoid “cryptographic disasters” due to insufficient cryptographic awareness of application developers. As such, cryptographic agility and configurability are considered “antifeatures” due to the inherent complexity costs and the risk of enabling adopters to select insecure combinations of primitives.

From this brief survey, PKCS #11, due to its widespread support, appears to be a suitable candidate to provide alternative crypto primitive implementations portable across different libraries. Unfortunately, OpenSSL does not natively support PKCS #11, but only through a third party ENGINE implementation. Nonetheless, this application of PKCS #11 seems to be an interesting direction for further research on this topic.

## VII. Conclusion

In this work, we analyzed issues affecting real-world security across currently deployed, ubiquitous cryptography software libraries. The consequences derived from these issues include: (1) limited assurance that software implementations are functionally correct; (2) low or non-existent accountability in the software engineering decision-making process; (3) costly and nonscalable upkeep of custom builds augmented with new features; (4) and additional security risks due to conflicting release strategies between cryptographic libraries and OS vendors.

In addition, we also examined some factors that hinder researchers and practitioners from achieving timely and widespread dissemination of their scientific results in real-world applications, thus impeding the potential impact of current cryptography research.

As a possible solution, we presented the adoption of OpenSSL ENGINEs as a framework to overcome these limits and as a convenient tool for researchers to disseminate, test and benchmark their contributions in real-world applications. We demonstrated the usage, viability, limits and benefits of this framework through libsuola, a project that aims at providing support for emerging cryptography standards such as X25519 and Ed25519 for currently deployed versions of OpenSSL, while being completely transparent for existing applications.

Due to the nature of this research, we expect it to be the foundation for future work aiming at bridging the gaps between research results and real-world applications. Our methodology has already been applied [47] in NIST’s ongoing post-quantum cryptosystem standardization competition [48]. We intend to explore automated ENGINE construction, potentially leading to tooling that rigs together popular research-driven APIs (e.g. SUPERCOP) with practice-driven APIs. Moreover, we also expect it to serve as a means to enhance future research efforts, by providing a framework to ease the testing and benchmarking of novel scientific results in real-world systems and settings.

### *Acknowledgments*

Supported in part by Academy of Finland grant 303814.

This article is based in part upon work from COST Action IC1403 CRYPTACUS, supported by COST (European Cooperation in Science and Technology).

[16] https://www.gnutls.org/

[17] https://tls.mbed.org/

[18] https://tls.mbed.org/kb/development/hw_acc_guidelines

[19] https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

[20] https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

# References

[1] D. J. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” in *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, ser. Lecture Notes in Computer Science, A. Hevia and G. Neven, Eds., vol. 7533. Springer, 2012, pp. 159–176. [Online]. Available: https://doi.org/10.1007/978-3-642-33481-8_9

[2] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, “HACL*: A verified modern cryptographic library,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1789–1806. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134043

[3] D. J. Bernstein and T. Lange, “eBACS: ECRYPT Benchmarking of Cryptographic Systems,” [Online; accessed 30-August-2017]. [Online]. Available: https://bench.cr.yp.to

[4] D. J. Bernstein, “Curve25519: New Diffie-Hellman speed records,” in *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, ser. Lecture Notes in Computer Science, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., vol. 3958. Springer, 2006, pp. 207–228. [Online]. Available: https://doi.org/10.1007/11745853_14

[5] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B. Yang, “High-speed high-security signatures,” *J. Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, 2012. [Online]. Available: https://doi.org/10.1007/s13389-012-0027-1

[6] A. Langley, M. Hamburg, and S. Turner, “Elliptic Curves for Security,” Internet Requests for Comments, RFC Editor, RFC 7748, Jan. 2016. [Online]. Available: https://datatracker.ietf.org/doc/rfc7748/

[7] S. Josefsson and I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA),” Internet Requests for Comments, RFC Editor, RFC 8032, Jan. 2017. [Online]. Available: https://datatracker.ietf.org/doc/rfc8032/

[8] E. Rescorla, “The transport layer security (TLS) protocol version 1.3,” Internet Requests for Comments, RFC Editor, RFC 8446, Aug. 2018. [Online]. Available: https://datatracker.ietf.org/doc/rfc8446/

[9] Y. Nir, S. Josefsson, and M. Pégourié-Gonnard, “Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier,” Internet Requests for Comments, RFC Editor, RFC 8422, Aug. 2018. [Online]. Available: https://datatracker.ietf.org/doc/rfc8422/

[10] E. Käsper and P. Schwabe, “Faster and timing-attack resistant AES-GCM,” in *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds., vol. 5747. Springer, 2009, pp. 1–17. [Online]. Available: https://doi.org/10.1007/978-3-642-04138-9_1

[11] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, “Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1006–1018. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978425

[12] C. Meyer and J. Schwenk, “SoK: Lessons learned from SSL/TLS attacks,” in *Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, ser. Lecture Notes in Computer Science, Y. Kim, H. Lee, and A. Perrig, Eds., vol. 8267. Springer, 2013, pp. 189–209. [Online]. Available: https://doi.org/10.1007/978-3-319-05149-9_12

[13] E. Biham, Y. Carmeli, and A. Shamir, “Bug attacks,” in *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, ser. Lecture Notes in Computer Science, D. A. Wagner, Ed., vol. 5157. Springer, 2008, pp. 221–240. [Online]. Available: https://doi.org/10.1007/978-3-540-85174-5_13

[14] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, “Practical realisation and elimination of an ECC-related software bug attack,” in *Topics in Cryptology - CT-RSA 2012 - The Cryptographers’ Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, ser. Lecture Notes in Computer Science, O. Dunkelman, Ed., vol. 7178. Springer, 2012, pp. 171–186. [Online]. Available: https://doi.org/10.1007/978-3-642-27954-6_11

[15] R. Dubois, “Trapping ECC with invalid curve bug attacks,” *IACR Cryptology ePrint Archive*, vol. 2017, no. 554, 2017. [Online]. Available: http://eprint.iacr.org/2017/554

[16] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, “Jasmin: High-assurance and high-speed cryptography,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 1807–1823. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134078

[17] C. Percival, “Cache missing for fun and profit,” in *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. [Online]. Available: http://www.daemonology.net/papers/cachemissing.pdf

[18] D. J. Bernstein, “Cache-timing attacks on AES,” 2005. [Online]. Available: http://cr.yp.to/papers.html#cachetiming

[19] S. Gueron, “Efficient software implementations of modular exponentiation,” *J. Cryptographic Engineering*, vol. 2, no. 1, pp. 31–43, 2012. [Online]. Available: https://doi.org/10.1007/s13389-012-0031-5

[20] D. J. Bernstein and P. Schwabe, “A word of warning,” August 2013, cHES 2013 Rump Session. [Online]. Available: https://cryptojedi.org/peter/data/chesrump-20130822.pdf

[21] Y. Yarom, D. Genkin, and N. Heninger, “CacheBleed: A timing attack on OpenSSL constant time RSA,” in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds., vol. 9813. Springer, 2016, pp. 346–367. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_17

[22] C. Pereida García, B. B. Brumley, and Y. Yarom, ““Make sure DSA signing exponentiations really are constant-time”,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 1639–1650. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978420

[23] B. B. Brumley and R. M. Hakala, “Cache-timing template attacks,” in *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, ser. Lecture Notes in Computer Science, M. Matsui, Ed., vol. 5912. Springer, 2009, pp. 667–684. [Online]. Available: https://doi.org/10.1007/978-3-642-10366-7_39

[24] E. Käsper, “Fast elliptic curve cryptography in OpenSSL,” in *Financial Cryptography and Data Security - FC 2011 Workshops, RLCPS and WECSR 2011, Rodney Bay, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. Danezis, S. Dietrich, and K. Sako, Eds., vol. 7126. Springer, 2011, pp. 27–39. [Online]. Available: https://doi.org/10.1007/978-3-642-29889-9_4

[25] S. Gueron and V. Krasnov, “Fast prime field elliptic-curve cryptography with 256-bit primes,” *J. Cryptographic Engineering*, vol. 5, no. 2, pp. 141–151, 2015. [Online]. Available: https://doi.org/10.1007/s13389-014-0090-x

[26] B. B. Brumley, “Faster software for fast endomorphisms,” in *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015, Berlin, Germany, April 13-14, 2015. Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Mangard and A. Y. Poschmann, Eds., vol. 9064. Springer, 2015, pp. 127–140. [Online]. Available: https://doi.org/10.1007/978-3-319-21476-4_9

[27] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, ““Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way,” in *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. Batina and M. Robshaw, Eds., vol. 8731. Springer, 2014, pp. 75–92. [Online]. Available: https://doi.org/10.1007/978-3-662-44709-3_5

[28] J. van de Pol, N. P. Smart, and Y. Yarom, “Just a little bit more,” in *Topics in Cryptology - CT-RSA 2015, The Cryptographer’s*

*Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings*, ser. Lecture Notes in Computer Science, K. Nyberg, Ed., vol. 9048. Springer, 2015, pp. 3–21. [Online]. Available: https://doi.org/10.1007/978-3-319-16715-2_1

[29] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, S. Schwab, W. K. Robertson, and D. Balzarotti, Eds. ACM, 2016, pp. 422–435. [Online]. Available: http://doi.acm.org/10.1145/2991079.2991084

[30] N. Tuveri, S. ul Hassan, C. Pereida García, and B. B. Brumley, "Side-channel analysis of SM2: A late-stage featurization case study," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 147–160. [Online]. Available: https://doi.org/10.1145/3274694.3274725

[31] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings, 20-22 May 2019, San Francisco, California, USA*. IEEE, 2019, pp. 1036–1053. [Online]. Available: https://doi.org/10.1109/SP.2019.00066

[32] D. J. Bernstein and P. Schwabe, "NEON crypto," in *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, ser. Lecture Notes in Computer Science, E. Prouff and P. Schaumont, Eds., vol. 7428. Springer, 2012, pp. 320–339. [Online]. Available: https://doi.org/10.1007/978-3-642-33027-8_19

[33] O. P. . TC, "PKCS #11 Cryptographic Token Interface Base Specification Version 2.40 Plus Errata 01," OASIS Standard, OASIS, Organization for the Advancement of Structured Information Standards, Standard incorporating approved Errata, May 2016. [Online]. Available: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/errata01/os/pkcs11-base-v2.40-errata01-os-complete.html

[34] J. Linn, "Generic Security Service Application Program Interface, Version 2," Internet Requests for Comments, RFC Editor, RFC 2078, Jan. 1997. [Online]. Available: https://datatracker.ietf.org/doc/rfc2078/

[35] ——, "Generic Security Service Application Program Interface Version 2, Update 1," Internet Requests for Comments, RFC Editor, RFC 2743, Jan. 2000. [Online]. Available: https://datatracker.ietf.org/doc/rfc2743/

[36] D. C. Adams, "Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API)," Internet Requests for Comments, RFC Editor, RFC 2479, Dec. 1998. [Online]. Available: https://datatracker.ietf.org/doc/rfc2479/

[37] T. O. Group, "Common Data Security Architecture (CDSA), Version 2," May 2000.

[38] N. C.-O. C. Team, "Security service api: Cryptographic api recommendation," 1996.

[39] V. Smyslov, "Simple Cryptographic Program Interface (Crypto API)," Internet Requests for Comments, RFC Editor, RFC 2628, Jun. 1999. [Online]. Available: https://datatracker.ietf.org/doc/rfc2628/

[40] A. D. Keromytis, J. L. Wright, and T. de Raadt, "The design of the OpenBSD cryptographic framework," in *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*. USENIX, 2003, pp. 181–196. [Online]. Available: http://www.usenix.org/events/usenix03/tech/keromytis.html

[41] A. D. Keromytis, J. L. Wright, T. de Raadt, and M. Burnside, "Cryptography as an operating system service: A case study," *ACM Trans. Comput. Syst.*, vol. 24, no. 1, pp. 1–38, 2006. [Online]. Available: http://doi.acm.org/10.1145/1124153.1124154

[42] Y. Dutta and V. Sethi, "Performance analysis of cryptographic acceleration in multicore environment," in *Quality, Reliability, Security and Robustness in Heterogeneous Networks - 9th International Conference, QShine 2013, Greader Noida, India, January 11-12, 2013, Revised Selected Papers*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, K. Singh and A. K. Awasthi, Eds., vol. 115. Springer, 2013, pp. 658–667. [Online]. Available: https://doi.org/10.1007/978-3-642-37949-9_57

[43] N. Mavrogiannopoulos, M. Trmac, and B. Preneel, "A linux kernel cryptographic framework: decoupling cryptographic keys from applications," in *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, S. Ossowski and P. Lecca, Eds. ACM, 2012, pp. 1435–1442. [Online]. Available: http://doi.acm.org/10.1145/2245276.2232006

[44] U.S. National Security Agency, "NSA Suite B Cryptography," January 2009. [Online]. Available: http://www.nsa.gov/ia/programs/suiteb_cryptography/

[45] P. Gutmann, "cryptlib security toolkit, 3.4.3.1," January 2017. [Online]. Available: https://www.cs.auckland.ac.nz/~pgut001/cryptlib/

[46] ——, *Cryptographic Security Architecture. Design and Verification*, 1st ed. Springer-Verlag New York, 2004.

[47] A. Barnett and A. Byrne, "Lattice-based cryptographic key management prototype," H2020: Secure Architectures of Future Emerging Cryptography (SAFEcrypto), Deliverable D8.3. [Online]. Available: https://cordis.europa.eu/project/rcn/194240/results/en

[48] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," National Institute of Standards and Technology, NISTIR 8105, April 2016. [Online]. Available: http://doi.org/10.6028/NIST.IR.8105

## Appendix A
## Detailed benchmark results

TABLE III
Benchmark results on a 4-cores/8-threads Intel Core i5-6700 CPU (Skylake) running at 3.4GHz, with Enhanced Intel SpeedStep Technology and Intel Turbo Boost Technology disabled. The workstation is equipped with 32GB 2667MHz DRAM, and runs Ubuntu 18.04.2 LTS with Linux x86_64 kernel 4.15.0-45-generic. For each operation we collected 12800 samples and computed the median value. In parenthesis the CPU cycles ratio between the current column and the reference (leftmost) value.

| Operation | CPU cycles per operation<br>OpenSSL-1.0.2r | | | |
|---|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistz256 keygen | 41528 | — | — | — |
| X25519 keygen | — | 128581 | 90320 (1.4x) | 166736 (0.8x) |
| Ed25519 keygen | — | 80286 | 92710 (0.9x) | 300552 (0.3x) |
| nistz256 derive | 172436 | — | — | — |
| X25519 derive | — | 135892 | 149116 (0.9x) | 159182 (0.9x) |
| nistz256 sign | 118090 | — | — | — |
| Ed25519 sign | — | 82160 | 88426 (0.9x) | 587925 (0.1x) |
| nistz256 verify | 253644 | — | — | — |
| Ed25519 verify | — | 207832 | 284916 (0.7x) | 611872 (0.3x) |
| Operation | CPU cycles per operation<br>OpenSSL-1.1.0j | | | |
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistz256 keygen | 43542 | — | — | — |
| X25519 keygen | 117484 | 129992 (0.9x) | 91510 (1.3x) | 167815 (0.7x) |
| Ed25519 keygen | — | 80250 | 92924 (0.9x) | 301606 (0.3x) |
| nistz256 derive | 172404 | — | — | — |
| X25519 derive | 342231 | 136888 (2.5x) | 149159 (2.3x) | 159084 (2.2x) |
| nistz256 sign | 124082 | — | — | — |
| Ed25519 sign | — | 82161 | 87204 (0.9x) | 587762 (0.1x) |
| nistz256 verify | 259612 | — | — | — |
| Ed25519 verify | — | 205114 | 286386 (0.7x) | 611105 (0.3x) |
| Operation | CPU cycles per operation<br>OpenSSL-1.1.1b | | | |
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistz256 keygen | 38877 | — | — | — |
| X25519 keygen | 114124 | 128595 (0.9x) | 86838 (1.3x) | 162968 (0.7x) |
| Ed25519 keygen | 115230 | 80232 (1.4x) | 88382 (1.3x) | 297225 (0.4x) |
| nistz256 derive | 171662 | — | — | — |
| X25519 derive | 118312 | 135398 (0.9x) | 149148 (0.8x) | 159346 (0.7x) |
| nistz256 sign | 75490 | — | — | — |
| Ed25519 sign | 114782 | 82287 (1.4x) | 87428 (1.3x) | 587884 (0.2x) |
| nistz256 verify | 231329 | — | — | — |
| Ed25519 verify | 374236 | 207457 (1.8x) | 284330 (1.3x) | 611566 (0.6x) |

TABLE IV

Benchmark results on a Raspberry Pi 3B, equipped with a quad-core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB RAM, running Ubuntu 18.04.2 LTS and 32-bit armv7l Linux kernel version 4.15.0-1031-raspi2. A CPU core was reserved exclusively for the benchmark process, and frequency scaling disabled via software. We monitored the RTOS running on the embedded VideoCore VPU controlling the SoC to ensure that undervoltage, frequency capping and external throttling were avoided for the duration of the benchmark. For each operation we collected 12800 samples and computed the median value. In parenthesis the CPU cycles ratio between the current column and the reference (leftmost) value.

| Operation | CPU cycles per operation<br>OpenSSL-1.0.2r | | |
|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna |
| nistp256 keygen | 5915353 | — | — |
| X25519 keygen | — | 582951 | 506892 (1.2x) |
| Ed25519 keygen | — | 593302 | 513263 (1.2x) |
| nistp256 derive | 6054496 | — | — |
| X25519 derive | — | 1569018 | 484920 (3.2x) |
| nistp256 sign | 6320842 | — | — |
| Ed25519 sign | — | 616191 | 503163 (1.2x) |
| nistp256 verify | 4155613 | — | — |
| Ed25519 verify | — | 1766874 | 1509083 (1.2x) |

| Operation | CPU cycles per operation<br>OpenSSL-1.1.0j | | |
|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna |
| nistz256 keygen | 219973 | — | — |
| X25519 keygen | 575106 | 583914 (1.0x) | 510528 (1.1x) |
| Ed25519 keygen | — | 595987 | 516408 (1.2x) |
| nistz256 derive | 1149047 | — | — |
| X25519 derive | 1546213 | 1568875 (1.0x) | 484890 (3.2x) |
| nistz256 sign | 553960 | — | — |
| Ed25519 sign | — | 618496 | 502348 (1.2x) |
| nistz256 verify | 1503140 | — | — |
| Ed25519 verify | — | 1771464 | 1508409 (1.2x) |

| Operation | CPU cycles per operation<br>OpenSSL-1.1.1b | | |
|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna |
| nistz256 keygen | 221288 | — | — |
| X25519 keygen | 582162 | 583748 (1.0x) | 510915 (1.1x) |
| Ed25519 keygen | 587208 | 596005 (1.0x) | 516872 (1.1x) |
| nistz256 derive | 1147024 | — | — |
| X25519 derive | 1550240 | 1568880 (1.0x) | 484880 (3.2x) |
| nistz256 sign | 515348 | — | — |
| Ed25519 sign | 576812 | 621402 (0.9x) | 505244 (1.1x) |
| nistz256 verify | 1559354 | — | — |
| Ed25519 verify | 1757511 | 1792064 (1.0x) | 1530556 (1.1x) |

TABLE V

Benchmark results on a Raspberry Pi 3B, equipped with a quad-core 1.2GHz Broadcom BCM2837 64bit CPU and 1GB RAM, running Ubuntu 18.04.2 LTS and 64-bit aarch64 Linux kernel version 4.15.0-1031-raspi2. A CPU core was reserved exclusively for the benchmark process, and frequency scaling disabled via software. We monitored the RTOS running on the embedded VideoCore VPU controlling the SoC to ensure that undervoltage, frequency capping and external throttling were avoided for the duration of the benchmark. For each operation we collected 12800 samples and computed the median value. In parenthesis the CPU cycles ratio between the current column and the reference (leftmost) value.

| Operation | CPU cycles per operation<br>OpenSSL-1.0.2r | | | |
|---|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistp256 keygen | 401806 | — | — | — |
| X25519 keygen | — | 213138 | 182785 (1.2x) | 524648 (0.4x) |
| Ed25519 keygen | — | 216960 | 186450 (1.2x) | 993557 (0.2x) |
| nistp256 derive | 1087916 | — | — | — |
| X25519 derive | — | 558776 | 479496 (1.2x) | 514212 (1.1x) |
| nistp256 sign | 808516 | — | — | — |
| Ed25519 sign | — | 221356 | 174227 (1.3x) | 1956526 (0.1x) |
| nistp256 verify | 1525158 | — | — | — |
| Ed25519 verify | — | 656302 | 566792 (1.2x) | 2027707 (0.3x) |

| Operation | CPU cycles per operation<br>OpenSSL-1.1.0j | | | |
|---|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistz256 keygen | 140707 | — | — | — |
| X25519 keygen | 242538 | 213084 (1.1x) | 184170 (1.3x) | 526310 (0.5x) |
| Ed25519 keygen | — | 217356 | 187964 (1.2x) | 995222 (0.2x) |
| nistz256 derive | 539104 | — | — | — |
| X25519 derive | 536012 | 558572 (1.0x) | 479776 (1.1x) | 514541 (1.0x) |
| nistz256 sign | 328727 | — | — | — |
| Ed25519 sign | — | 221468 | 174797 (1.3x) | 1956594 (0.1x) |
| nistz256 verify | 792487 | — | — | — |
| Ed25519 verify | — | 652126 | 566749 (1.2x) | 2027784 (0.3x) |

| Operation | CPU cycles per operation<br>OpenSSL-1.1.1b | | | |
|---|---|---|---|---|
| | default | libsuola-sodium | libsuola-donna | libsuola-hacl |
| nistz256 keygen | 144158 | — | — | — |
| X25519 keygen | 250089 | 212957 (1.2x) | 187706 (1.3x) | 530116 (0.5x) |
| Ed25519 keygen | 253400 | 217181 (1.2x) | 191261 (1.3x) | 998444 (0.3x) |
| nistz256 derive | 539346 | — | — | — |
| X25519 derive | 519758 | 558584 (0.9x) | 479632 (1.1x) | 514342 (1.0x) |
| nistz256 sign | 238736 | — | — | — |
| Ed25519 sign | 235394 | 221840 (1.1x) | 174794 (1.3x) | 1956938 (0.1x) |
| nistz256 verify | 721034 | — | — | — |
| Ed25519 verify | 619356 | 648358 (1.0x) | 563364 (1.1x) | 2028430 (0.3x) |