

Randomness-Anchored Runtime Discovery of Cryptographic Operations on Linux Using eBPF

Sumin Jeong¹[0009–0007–9643–4872], Yulim Hyoung¹[0009–0009–7530–6497],
Hagyeong Kim¹[0009–0008–5840–5517], Huiju Kang¹[0009–0008–3198–4598], and
HwaJeong Seo¹[0000–0003–0069–9061]

Hansung University, Seoul, South Korea
jeong9sumin@gmail.com yulim4hyoung@gmail.com
kimhaha4420@gmail.com huiju9190@gmail.com
hwajeong84@gmail.com

Abstract. Migration to post-quantum cryptography requires knowing not only which cryptographic libraries are *present* on a system, but which algorithms are *actually executed* at runtime. Static binary analysis answers the former question; the latter requires dynamic observation. We present a runtime cryptographic discovery system for Linux, built on eBPF user-space probes, that detects classical and post-quantum operations as they execute. Because key generation and encapsulation consume fresh randomness, the system treats randomness observations as temporal *anchors* and correlates them with the cryptographic calls that follow in the same process, producing a stream of confidence-scored events for a runtime cryptographic inventory. An ablation over three detection configurations shows that randomness alone is not a sound detector, while API detection reaches perfect precision and recall on classical and ML-KEM workloads with no false positives on random-only negatives, and the randomness anchor raises the confidence of those detections without adding any. A single probe set also covers native post-quantum support in current OpenSSL releases. Median detection latency is $28\mu\text{s}$ (P99 $48\mu\text{s}$), largely insensitive to a four-round tuning sweep, with overhead of 14–18% on a key-generation-bound workload, full detection under sustained and concurrent load, and stable behavior over a one-hour run. A case study across nine application cases drawn from or paired with the QED datasets shows where static and runtime views agree and how probe coverage determines what runtime discovery can observe.

Keywords: Post-quantum cryptography · eBPF · Runtime detection · Randomness · Cryptographic bill of materials · Dynamic analysis.

1 Introduction

Standardization of post-quantum cryptography (PQC), including ML-KEM (FIPS 203) [26] and ML-DSA (FIPS 204) [25], has turned cryptographic migration into an operational problem: organizations must inventory where quantum-vulnerable primitives such as RSA and ECDH are used, and verify

where PQC has actually been adopted [13, 23, 28]. The emerging notion of a cryptographic bill of materials (CBOM) [30] formalizes this inventory in a machine-readable form that can be carried with a software bill of materials. Such an inventory identifies cryptographic algorithms and parameters, but it does not establish which of them are actually executed at runtime.

Cryptographic-inventory approaches for migration are predominantly *static*. QED (Quantum-vulnerable Executable Detection) [35] analyzes software executables and their dependencies to flag quantum-vulnerable cryptography before deployment. Static analysis is fast and requires no execution, but it primarily establishes presence: a binary may link a cryptographic library without invoking the relevant primitive in a given run. Conversely, cryptography reached through dynamic loading or indirect calls may escape scanners that rely on static dependency or import information. *Which* algorithm actually executes, *when* it executes, and *how often* are therefore runtime properties.

This paper studies dynamic discovery: identifying cryptographic operations as they execute, in real time and with low overhead. We use randomness as a temporal anchor because several operations evaluated in this work, including key generation, KEM encapsulation, randomized signing, and TLS handshakes, draw fresh randomness through a relatively small set of system or library RNG interfaces. When an RNG event is observed, the monitor opens a short per-process window and correlates subsequent cryptographic API events with that observation. The cryptographic API event remains the primary detection signal; the randomness observation adds temporal context and raises the event’s confidence. Randomness alone is insufficient as a detector because non-cryptographic activity such as UUID generation and process start-up can use the same randomness sources.

Our system is implemented with eBPF [5] user-space probes (uprobes) on OpenSSL 3 and liboqs [40], a kernel ring buffer, and a user-space daemon that emits JSON events suitable for CBOM construction. The contributions are:

1. **Randomness-anchored detection model.** We evaluate three runtime views: randomness activity (D_{random}), cryptographic API invocation (D_{api}), and API events annotated with a recent randomness observation (D_{corr}). The ablation shows why randomness alone is insufficient and quantifies the temporal context added by correlation without changing the API detector’s decisions.
2. **PQC visibility before and after native OpenSSL support.** When the evaluated OpenSSL version lacks native PQC support, we detect ML-KEM-512/768/1024 and ML-DSA-65 through liboqs uprobes. The generic OpenSSL EVP probe paths also observe OpenSSL 3.5 native ML-KEM and ML-DSA operations, and we validate this coverage on live TLS 1.3 handshakes using a PQC hybrid group.
3. **Performance characterization.** A four-round tuning study of the poll interval and output format shows a $28\mu\text{s}$ median detection latency (P99 $48\mu\text{s}$), with the observed tuning effect confined to the tail on our host. We additionally measure throughput, scalability, and one-hour behavior.

4. **Static-vs-runtime case study.** We compare static dependency evidence with runtime observations across nine application cases drawn from or paired with the QED datasets, spanning network tools, TPM2 tools, and a synthetic RSA workload. The case study also shows how a sign/verify-only workload depends on the probe manifest.

The rest of the paper is organized as follows. Section 2 surveys related work on cryptographic inventory and eBPF-based observability. Section 3 presents the three detection configurations and the randomness-anchoring model. Section 4 evaluates detection correctness, post-quantum coverage, performance, and the static-vs-runtime case study. Section 5 discusses the results and concludes.

2 Background and Related Work

2.1 Cryptographic Inventory for Post-Quantum Migration

Policy. Migration guidance from governments and industry places a cryptographic inventory at the start of the process. OMB Memorandum M-23-02 directs US federal agencies to inventory quantum-vulnerable cryptography [28], the joint CISA/NSA/NIST quantum-readiness factsheet names the same first step [3], and Joseph et al. describe organizational migration as an inventory-driven program [13]. In Korea, PQC standardization, migration policy, and crypto-agility are likewise under active discussion [46, 32].

Tooling. The NCCoE migration project [27] evaluates cryptographic-discovery tools for migration planning. Näther et al. [24] identify the lack of a concrete cryptographic inventory as a migration challenge, while Ott et al. [29] frame cryptographic agility as a broader research problem. CycloneDX CBOM [30] provides a schema for representing cryptographic assets, and tools such as IBM’s CBOMkit [11] can populate an inventory from source code and software artifacts. These static methods describe cryptographic assets present in source code, binaries, or dependencies, but they do not establish which operations execute in a particular run. Our runtime inventory complements them by recording cryptographic operations observed during execution.

2.2 Static Identification of Cryptography in Software

QED. QED [35] analyzes executables in three static phases, moving from file-level dependencies to API-level dependencies and then to static traces that reach quantum-vulnerable APIs. Its evaluation includes a synthetic dataset built from two OpenSSL versions, MbedTLS, and wolfSSL, together with a real-world dataset containing Coreutils, UnixBench, network utilities, and tpm2-tools. We use QED’s datasets as the basis for the static-vs-runtime case study in Sect. 4.5.

Primitive identification. A longer line of work identifies cryptographic primitives inside binary code without relying on symbols: Lestringant et al. match data-flow graph signatures [17], and Meijer et al. extend the idea to proprietary primitives [22]. At source level, CrySL [14] specifies and checks correct cryptographic-API usage, while CryptoGuard [34] statically detects cryptographic API misuse in Java programs. Static analysis can establish that cryptographic code or dependencies are present without executing the program, but presence does not show whether a primitive runs in a particular execution. Approaches that rely on import or dependency metadata may also miss cryptography reached through runtime loading or other indirect mechanisms.

2.3 Dynamic Identification of Cryptography

Dynamic identification observes a program as it runs. Gröbert et al. recover primitives from Pin-instrumented execution traces [8]; Aligot [2] and Crypto-Hunt [45] identify obfuscated implementations by matching loop input-output relations and bit-precise symbolic loop mappings; K-Hunt [18] pinpoints key material in traces; and Dispatcher [1] and ReFormat [44] locate decryption routines by the ratio of arithmetic and bitwise instructions. These systems are designed for targeted binary analysis rather than continuous inventory monitoring. They rely on dynamic binary instrumentation, emulation, execution traces, or symbolic analysis, giving them a different operating model from a monitor intended to remain active on production hosts. Our setting instead assumes that:

- the monitored cryptographic libraries and exported interfaces are known;
- the monitor runs continuously on production hosts; and
- each event should identify the cryptographic operation and, when the algorithm name can be resolved, its post-quantum status.

These assumptions motivate symbol-level uprobes rather than whole-program instruction tracing; Sect. 3 presents the design and Sect. 4.4 measures its runtime cost.

2.4 eBPF-Based Observability

Mechanism. eBPF runs verified programs inside the Linux kernel [6, 43], attached to kernel or user-space instrumentation points, with data exported through ring buffers [5, 19]. A *uprobe* [4] patches a breakpoint at a user-space symbol; on each call the attached program executes in kernel context, and a *uretprobe* does the same on return. Uprobes can be attached to running programs without modifying or recompiling the target, allowing selected user-space function calls to be observed without whole-program instrumentation.

TLS capture tools. Uprobes on OpenSSL are already used to capture TLS plaintext for debugging and traffic inspection, in the `sslsniff` tool of bcc [12], eCapture [7], and Pixie [33]. Those tools instrument TLS I/O paths

to recover plaintext or session-key material; our monitor instead instruments cryptographic-operation paths and records event metadata such as the operation type, timestamp, process identifier, symbol, and resolved algorithm name without copying TLS payloads or key material.

Privileges and scope. The reach of eBPF also carries risk: He et al. show how eBPF privileges can be abused across containers [9], which is why Sect. 5 assumes a trusted observer with the privileges required to load the monitor. Prior Korean work has applied eBPF to packet payload inspection and container-attack detection [47, 37]. A July 2026 IETF Internet-Draft on cryptographic asset discovery lists eBPF-based system-call auditing as one process-identification technique [21]. That guidance discusses eBPF at the process-identification level; our work instead evaluates algorithm-level user-space probes, randomness correlation, and post-quantum classification.

2.5 Randomness as a Signal

Prior uses. Randomness and key material are closely linked, and prior work has used the link in both directions. Prior work has used properties of randomness to locate or diagnose cryptographic key material. Shamir and van Someren locate stored keys using the high entropy of their bytes [36], while K-Hunt tracks random-number-generator outputs into the cryptographic routines that consume them to identify key material [18]. Heninger et al. show a related failure mode in which insufficient entropy during boot leads to widespread weak keys [10].

Randomness as an event. We use randomness in a different form, as an *event*: the moment a process asks the system or library generator for bytes [16, 20]. Key generation for RSA, elliptic-curve schemes, and ML-KEM all consume randomness, but they do not all take the same route to it:

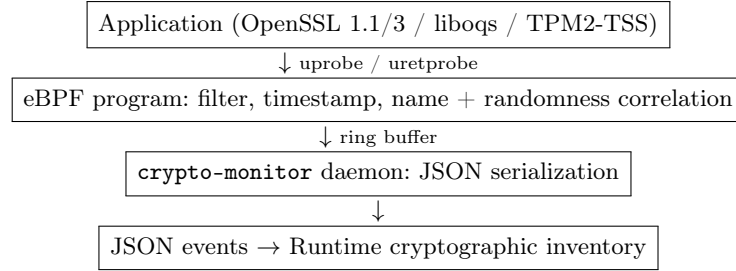
- in the OpenSSL 3 path observed here, key generation reaches the private generator through `RAND_priv_bytes` or `RAND_priv_bytes_ex`, rather than the public path behind `RAND_bytes`;
- in the liboqs build evaluated here, randomness reaches OpenSSL through `RAND_bytes`;
- an application may bypass both and ask the kernel directly.

In the implementations evaluated here, these RNG entry points are fewer than the algorithm-specific cryptographic interfaces. We therefore use RNG activity as a temporal anchor, not as an independent classifier of cryptographic execution. Sect. 3.4 explains why the relevant RNG path and probe position must be established for each library. Table 1 summarizes the scope of the approaches discussed above.

Table 1. Comparison with related approaches.

Approach	Executes	Invocation	Overhead	PQC	Metadata
Static binary [17, 22, 35]	no	no	none	QED only	n/a
Source scan [11, 14, 34]	no	no	none	partial	n/a
DBI dynamic [8, 18, 45]	yes	yes	high	no	no
eBPF TLS capture [7, 12, 33]	yes	TLS I/O	low	no	no
This work	yes	yes	low ($\leq 18\%$)	yes	yes

Executes: the target runs during analysis. *Invocation*: the approach observes that a primitive actually executes. *Overhead*: cost imposed on the target while it runs; for this work, the key-generation-bound worst case of Table 4. *PQC*: tells quantum-vulnerable from post-quantum algorithms. *Metadata*: no payload or key material is copied.

**Fig. 1.** System architecture.

3 Detection Model and Design

3.1 Architecture

Figure 1 shows the pipeline. Uprobes attach to cryptographic and RNG symbols in `libcrypto` (OpenSSL 3 and 1.1), `liboqs`, and `libtss2-esys` (TPM 2.0). The eBPF program filters events in kernel space, timestamps them, resolves the algorithm name, correlates each cryptographic event with the most recent randomness observation of the same process, assigns its confidence, and pushes fixed-size records into a ring buffer. The user-space daemon (`crypto-monitor`) polls the buffer and emits each record as a JSON event.

3.2 Event Model and Hooked Symbols

Each event carries a type, either a cryptographic operation or a randomness observation, together with the process id, a timestamp, the symbol that fired and, where it can be resolved, the algorithm name. The name enters at the constructor that receives the algorithm string and then follows the objects built from it, as described at the end of this subsection. Table 2 lists the symbols behind each role.

The monitor covers OpenSSL signing and verification through `EVP_PKEY_sign`, `EVP_PKEY_verify`, `EVP_DigestSignInit`, and `EVP_DigestVerifyInit`, together with the extended `_ex` initialization interfaces used by OpenSSL 3

Table 2. Instrumented libraries and representative symbols. “(uretprobe)”: the event is emitted when the call returns.

Library	Symbols (excerpt)	Coverage
libcrypto (OpenSSL 3)	EVP_PKEY_CTX_new_from_name, EVP_PKEY_keygen (uretprobe), EVP_PKEY_encapsulate, EVP_PKEY_decapsulate, EVP_PKEY_sign/verify, EVP_DigestSign/VerifyInit(_ex), EVP_PKEY_generate/fromdata, EVP_PKEY_CTX_new(_from_pkey), EVP_PKEY_CTX_free, RAND_bytes, RAND_bytes_ex, RAND_priv_bytes(_ex)	RSA, EC, X25519 ML-KEM, ML-DSA (OpenSSL 3.5) sign/verify, cert. verification name propagation RNG anchor
liboqs	OQS_KEM_new, OQS_KEM_keypair (uretprobe), OQS_KEM_encaps/decaps OQS_SIG_new, OQS_SIG_keypair (uretprobe), OQS_SIG_sign/verify	ML-KEM-512/768/1024 ML-DSA-65
libtss2-esys	Esys_CreatePrimary Esys_GetRandom	TPM 2.0 TPM RNG anchor
kernel	getrandom(2) tracepoint	syscall RNG anchor

applications and libssl. These probes also observe certificate-verification operations reached through the EVP layer during TLS handshakes. For liboqs, `OQS_SIG_new`, key-pair generation, signing, and verification cover the ML-DSA path, with the constructor providing the algorithm name as in ML-KEM. Sect. 4.3 evaluates these paths using ML-DSA-65 through both liboqs and the OpenSSL 3.5 native provider, as well as a TLS 1.3 handshake using `X25519MLKEM768` and a classical `X25519` control. The TLS workload requires no TLS-specific cryptographic probe because its operations reach the EVP interfaces listed above. OpenSSL 3.5 supports native ML-DSA message signing through the message-signing EVP interface and also permits `EVP_DigestSignInit_ex` and `EVP_DigestVerifyInit_ex` as compatibility paths when no external digest is selected; the probe set includes the latter interfaces to cover the command-line and libssl paths exercised in our experiments.

Name attribution follows the object that carries the name. A context constructed from an algorithm string (`EVP_PKEY_CTX_new_from_name`, `OQS_KEM_new`, `OQS_SIG_new`) is recorded with that name; The probe stores the name pointer at function entry and reads the string when the constructor returns, avoiding reliance on the user-space page being resident at the entry probe. A key produced through such a context, by key or parameter generation (`EVP_PKEY_generate`) or by import (`EVP_PKEY_fromdata` and the raw-key

constructors), inherits the context’s name, and a context later built from that key (`EVP_PKEY_CTX_new_from_pkey`, `EVP_PKEY_CTX_new`) inherits it in turn. This is how encapsulation, decapsulation, signing, and verification contexts, which the API requires to be created from a key rather than from a name, obtain their names. Name propagation is necessary because encapsulation, decapsulation, signing, and verification contexts are often created from keys rather than directly from algorithm names. The monitor therefore propagates the name from a context to a generated or imported key and then from that key to later contexts. It also removes the mapping when a context is freed so that a reused address cannot inherit a stale algorithm name. Freed contexts are forgotten, so a reused address cannot hand a stale name to a new object, and an event whose name cannot be resolved is reported with family *unknown* rather than classical. Hybrid group names can embed the PQC identifier after a classical component, as in `X25519MLKEM768`. The classifier therefore searches the full algorithm name for the MLKEM and MLDSA tokens instead of relying only on a prefix match.

3.3 Detection Configurations

We compare three detection configurations in Sect. 4.2:

- D_{random} : a process is flagged when randomness is observed through any monitored RNG interface.
- D_{api} : a process is flagged when it invokes a monitored cryptographic operation. This is the primary detection signal.
- D_{corr} : each D_{api} event is annotated with the most recent preceding RNG observation from the same process. If the gap Δt falls within the correlation window, the event is marked as anchored and its confidence score is increased. D_{corr} does not create detections beyond those produced by D_{api} ; it adds temporal context to an existing cryptographic event.

The correlation window is one second. For each cryptographic event, the monitor looks up the most recent randomness observation from the same process; if it occurred within the preceding 1 s, the event is marked as anchored and records both Δt and the number of random bytes observed. Correlation is maintained per process rather than per thread, allowing a draw in one thread to anchor a cryptographic call in another thread of the same process. A two-thread test confirms this behavior: a verification operation in a worker thread is associated with a draw made by the main thread approximately 200 ms earlier. Because process-level correlation can also associate independent activity from different threads, an anchor represents temporal proximity rather than proof that the cryptographic operation consumed the observed randomness. The one-second window is a conservative implementation choice rather than a tuned threshold. In the workloads reported in Sects. 4.2 and 4.5, all anchored events have Δt below 40 ms, so the evaluated data do not distinguish among thresholds between 40 ms and 1 s. For operations that consume randomness, such as key generation, encapsulation, and randomized signing, the anchor can correspond to a draw used

Algorithm 1 Emitting a cryptographic event in the eBPF program.

State: $Last[p]$, the time and size of the most recent draw by process p ; $CtxName$, $KeyName$, the names bound to contexts and keys (Sect. 3.2); window $W = 1$ s.

```

1: procedure ONRANDOMNESS( $p, b$ )                                ▷ any hooked generator
2:    $Last[p] \leftarrow (now, b)$ 
3: end procedure

4: procedure ONCRYPTO( $op, c, k$ )  ▷ context  $c$ , key  $k$ ; on return for key generation
5:    $e.op \leftarrow op$ ;  $e.\tau \leftarrow now$ 
6:    $e.name \leftarrow CtxName[c]$ , else  $KeyName[k]$ , else  $\perp$ 
7:   if  $Last[p]$  exists and  $e.\tau - Last[p].\tau \leq W$  then
8:      $e.anchored \leftarrow true$ ;  $e.\Delta t \leftarrow e.\tau - Last[p].\tau$ ;  $e.bytes \leftarrow Last[p].b$ 
9:   end if
10:   $e.family \leftarrow \text{CLASSIFY}(e.name)$  if  $e.name \neq \perp$ , else unknown
11:  if  $e.name = \perp$  then  $e.conf \leftarrow 0.5$ 
12:  else if  $e.anchored$  then  $e.conf \leftarrow 1.0$ 
13:  else  $e.conf \leftarrow 0.9$ 
14:  end if
15:  submit  $e$  to the ring buffer
16: end procedure

```

by the operation; for decapsulation and verification, it only places the event near recent randomness activity in the same process. Confidence is represented by a three-level heuristic score rather than a calibrated probability. An event whose algorithm name cannot be resolved receives 0.5, a named event receives 0.9, and a named event with a randomness anchor receives 1.0. An anchored event without a resolved name remains at 0.5 while retaining its anchor metadata, keeping name attribution and randomness correlation as separate signals. Algorithm 1 summarizes how the eBPF program builds each cryptographic event.

Because D_{corr} annotates events already produced by D_{api} rather than introducing a separate process-level decision rule, its precision and recall are identical to those of D_{api} by construction. We therefore evaluate the contribution of correlation using per-event anchor rate, Δt , and the heuristic confidence score.

Randomness alone is not sufficient evidence of a cryptographic operation, so D_{random} is used only as a contextual signal. D_{api} provides the primary detection signal, while D_{corr} annotates those detections with recent randomness without creating additional detections.

3.4 Where the Anchor Has to Be Read

Anchoring requires the relevant randomness observation to occur before the cryptographic event is emitted. This is not guaranteed when a probe is placed at function entry. For example, OpenSSL 3 draws RSA key-generation randomness from its private RNG path during the key-generation call itself, so an entry probe records the operation before the corresponding draw has been observed.

We therefore monitor the private RNG interfaces in addition to the public ones and emit key-generation events on function return. The same principle is applied to key-generation paths whose randomness is consumed internally before the call returns. Consequently, both the RNG entry point and the probe position are implementation-dependent properties that must be established for each monitored library. Sect. 4.2 evaluates the resulting correlation behavior.

3.5 Implementation

The kernel-side program and the user-space daemon are built with libbpf [19] against a Linux 6.x kernel with BTF, and target the OpenSSL 3 [41], liboqs [40], and TPM2-TSS [42] interfaces described above. The daemon resolves symbols in the target shared libraries and attaches the corresponding probes. For `libcrypto`, it attempts attachment to both the OpenSSL 3 and OpenSSL 1.1 libraries so that applications linked against either version can be observed; symbols unavailable in a particular version are skipped. An optional process-name filter is used in the real-world case study to suppress events from unrelated processes. Events are serialized as JSON in either the standard or compact output format, and Sect. 4.4 evaluates the output format together with the polling configuration. The WSL2 experiments run the monitor inside a privileged Docker container. The accompanying repository contains the prototype, workloads, measurement scripts, and summary logs used for the experiments reported in this paper.¹

4 Evaluation

4.1 Experimental Setup

Experiments ran on an AMD Ryzen 7 9800X3D under a BTF-enabled Linux 6.6 kernel (WSL2), with the monitor in a privileged container.² Workloads cover four groups:

- *Classical*: key generation for RSA, EC, and X25519.
- *Post-quantum through liboqs*: ML-KEM-512/768/1024 key pair, encapsulation, and decapsulation, and ML-DSA-65 key generation, signing, and verification.
- *Random-only negatives*: `getrandom`, `/dev/urandom` reads, and UUID generation, which exercise the RNG without any cryptographic operation.
- *OpenSSL 3.5*: EVP programs and the stock `openssl` command for native ML-KEM and ML-DSA, a TLS 1.3 handshake between `openssl s_server` and `s_client` with the hybrid group `X25519MLKEM768` and with classical `X25519`, and `curl` against a public HTTPS site.

¹ Source code, workloads, measurement scripts, and the summary logs: <https://github.com/yulim4hyoung/ebpf-pqc-runtime-discovery>.

² Ubuntu 24.04 container with its stock OpenSSL 3.0 and liboqs; Sect. 4.3 additionally bind-mount the host’s OpenSSL 3.5 `libcrypto` and `libssl` over the container’s copies.

Table 3. Detector ablation over 90 processes (six positive and three random-only workloads, ten runs each), without and with the `getrandom` tracepoint. P: precision; R: recall.

Detector	No tracepoint				Tracepoint attached			
	P	R	F_1	FP	P	R	F_1	FP
D_{random} (randomness alone)	1.00	1.00	1.00	0	0.67	1.00	0.80	30
D_{api} (cryptographic API)	1.00	1.00	1.00	0	1.00	1.00	1.00	0
D_{corr} (API + randomness anchor)	1.00	1.00	1.00	0	1.00	1.00	1.00	0

Each of the nine accuracy workloads (six positive, three negative) is run ten times as separate processes, with ground-truth labels recorded by the harness.

4.2 Detection Correctness and Detector Ablation

All classical and ML-KEM workloads are detected with precision, recall, and F_1 of 1.0, and all random-only negatives produce zero false positives. Table 3 shows the per-detector ablation over 90 processes, with the `getrandom` syscall tracepoint unavailable and attached. The D_{random} row depends on the vantage point: without the tracepoint the random-only negatives reach the kernel directly and bypass the hooked user-space generators, so randomness alone scores perfectly; once the tracepoint is attached, all 30 negative runs are flagged and D_{random} falls to precision 0.67 (F_1 0.80) while D_{api} and D_{corr} stay at 1.0 (Table 3, right; Sect. 5).

The real-world study below adds the same lesson from another angle: `ssh` and `tpm2_getrandom` emit hooked RNG events with no cryptographic operation, where D_{random} alone would raise false positives. This is the empirical basis for treating D_{random} as a contextual signal rather than the primary detector in Sect. 3.3.

What correlation adds is visible per event rather than per process. All 120 cryptographic events of the positive runs are named and anchored at confidence 1.0. The median Δt is 1.7 ms for RSA, whose last draw is followed by the rest of the key computation before the call returns (Sect. 3.4), and 53–88 μs for EC, X25519, and the three ML-KEM parameter sets, with a maximum of 2.0 ms. The ML-KEM key-pair events are anchored only because they are emitted on return, as the RSA key generation is; recorded at entry they would precede the draw they consume. In the real-world set of Sect. 4.5, 24 of the 26 cryptographic events are anchored, with Δt between 28 μs and 40 ms and all inside the window. Names are rarer. The name-propagation mechanism of Sect. 3 resolves algorithm names for five of the 26 real-world cryptographic events in Table 5. All five are anchored and receive the heuristic confidence score 1.0: curl’s X25519 key-generation event and the HMAC-related events from the two `tpm2_createprimary` runs. The other 21 belong to `wget`, built against OpenSSL 1.1, whose API has no constructor that takes a name, and to keys loaded from files or certificates (the synthetic RSA binary’s hard-coded key, TLS certificate verification), which carry no type

string. Name attribution, not correlation, is therefore what limits confidence on real traffic.

4.3 Post-Quantum Coverage

We check the probe set against three post-quantum routes: native ML-KEM and ML-DSA in OpenSSL 3.5, each alongside its liboqs counterpart, and a live TLS 1.3 handshake with a hybrid group. In the configurations evaluated here, liboqs provides the PQC path when the installed OpenSSL version lacks native ML-KEM and ML-DSA support, while the generic EVP probes also observe the native OpenSSL 3.5 implementations without PQC-specific probes.

Native ML-KEM and liboqs. OpenSSL 3.5 ships ML-KEM natively, so we checked whether the EVP probes that detect classical key generation also detect it. They do, and no ML-KEM-specific probe is involved: detection comes from the generic key-generation, encapsulation, and decapsulation symbols, and naming from the algorithm-independent propagation of Sect. 3. We ran two workloads against the host’s OpenSSL 3.5: a small EVP program that generates an ML-KEM key pair and then encapsulates and decapsulates, and the stock `openssl genpkey` command. In both cases every operation emits an API event. The key-generation events carry the algorithm name, recovered by the context-naming probe of Sect. 3, are classified as post-quantum, and follow a fresh randomness draw, reaching full confidence. The encapsulation and decapsulation contexts are created from the generated key rather than from a name; through the key-to-context propagation of Sect. 3 they inherit the key’s name, so all fifteen events of the EVP program (three key generations and twelve encapsulations and decapsulations across ML-KEM-512, 768, and 1024) are named, classified as post-quantum, and emitted at full confidence, and no name is carried over from one parameter set to the next. Each parameter set contributes two encapsulation and two decapsulation events because the program, like most EVP callers, first calls each function with a null output buffer to query the output lengths and then calls it again to perform the operation, and both calls reach the hooked symbol. The liboqs probes name all three operations directly, because the constructor there receives the algorithm string.

Native ML-DSA and liboqs. The ML-DSA results follow the same pattern as the ML-KEM results. Through liboqs, ML-DSA-65 key generation, signing, and verification are all detected and named at full confidence, the signature being drawn after a fresh randomness event (OQS_SIG keypair, sign, and verify; signature length 3309 B). On the native OpenSSL 3.5 path, an EVP program that generates a key and then signs and verifies through the message-signing interface that 3.5 introduces yields, for each of ML-DSA-44, 65, and 87, a key-generation event, two signing events (the size query and the signature, counted as for ML-KEM above), and a verification event; all twelve carry their algorithm

name through the key-to-context propagation of Sect. 3, are classified as post-quantum, and are anchored at full confidence. The stock `openssl genpkey` and `pkeyutl` commands of the same release yield a named key-generation event and one signing and one verification event that are detected and anchored but carry no name, because `pkeyutl` decodes its key from a file written by another process rather than generating it, the decoded-key gap listed among the future work.

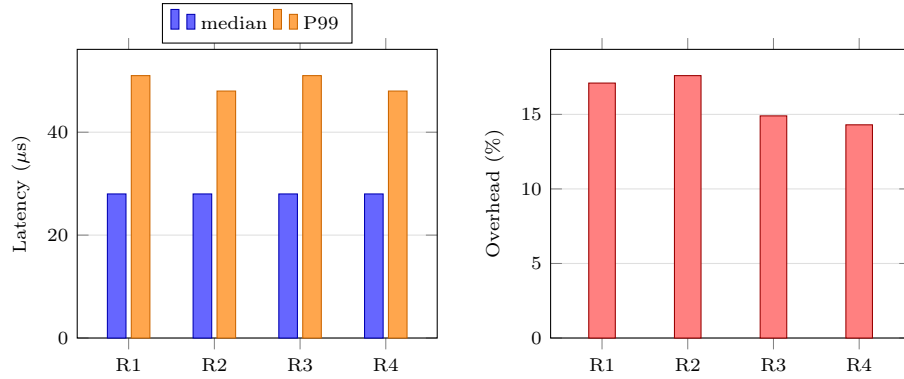
Real TLS 1.3 handshakes. Beyond synthetic workloads, we monitored a live TLS 1.3 handshake between `openssl s_server` and `s_client`, comparing a PQC hybrid group (`X25519MLKEM768`, the hybrid key agreement standardized for TLS 1.3 [15]) against a classical group (`X25519`). Post-quantum TLS handshakes have been benchmarked extensively [31, 38, 39]; our interest is in whether the host-side monitor attributes them correctly. The hybrid handshake produces eleven cryptographic events, all of which are anchored. Eight events are named: seven are attributed to `X25519MLKEM768` or `ML-KEM-768`, and one to the classical `X25519` component. The remaining three are signature-related initialization events whose algorithm names are not resolved and therefore retain heuristic confidence 0.5. The classical `X25519` control produces five cryptographic events, also all anchored: two named `X25519` key-generation events and three unnamed signature-related events. Thus, the hybrid run is distinguished by the additional ML-KEM-related operations and post-quantum attribution, while the classical run contains no post-quantum event. We also evaluate the name-propagation mechanism with a real client. a `curl` request to a public HTTPS site negotiated `X25519MLKEM768` and its key generation and decapsulations were attributed and anchored at full confidence, while the three certificate-verification events stayed unnamed, because certificate keys are decoded without a type string. The same request from the QED `curl` binary of Table 5, which runs against the container’s OpenSSL 3.0, negotiates classical `X25519` and yields four events (one key generation and three verifications); against OpenSSL 3.5 the client negotiates the hybrid group and yields eight (two key generations, three decapsulations, and three verifications). The runtime view thus reports the group actually negotiated, which a dependency inventory cannot. The hybrid group is classified as post-quantum using the same token-matching rule described in Sect. 3, while classical `X25519` remains classified as classical.

4.4 Performance

Four-round tuning. Detection latency is the time from the instrumented call to the daemon emitting the event. We tuned two parameters over four rounds: ring-buffer poll interval and JSON verbosity (Table 4, Fig. 2). The poll interval is the timeout of the daemon’s blocking wait on the ring buffer, not a sampling period: the daemon wakes as soon as a record is submitted, and the interval only bounds how long it sleeps when none is, which is why a 100 ms interval is compatible with a median latency of tens of microseconds. Overhead is the difference in workload completion time against an unmonitored baseline.

Table 4. Tuning rounds. Overhead is measured on a long-run RSA-2048 key-generation workload (500 iterations); see Sect. 5.

Round	Poll (ms)	Compact JSON	Median (μ s)	P95	P99	Overhead
R1 baseline	100	no	28	39	51	+17.1 %
R2 fast poll	10	no	28	35	48	+17.6 %
R3 aggressive	1	no	28	36	51	+14.9 %
R4 minimal output	1	yes	28	35	48	+14.3 %

**Fig. 2.** Latency and overhead across the four tuning rounds.

On this host, the two tuning parameters have little effect on the median, which remains 28 μ s in all four rounds, indicating that the BPF-to-user-space hand-off dominates end-to-end latency rather than the poll interval or serialization cost, consistent with the kernel-user transition cost reported for uprobes elsewhere [48]. Their effect is more visible in the tail, but the measurements below 10 ms do not show a consistent monotonic trend. Shortening the poll interval from 100 to 10 ms tightens P95 and P99 from 39 and 51 μ s to 35 and 48 μ s, the tightest distribution we measured, and the effect saturates there: R4, at 1 ms with compact output, reproduces the same 35 and 48 μ s, while R3, at 1 ms with verbose output, returns to a P99 of 51 μ s. Below 10 ms neither the poll interval nor the output format moves the tail in a consistent direction (Fig. 3). The CDF also has a lower mode: about one event in six is delivered within 1–25 μ s. These events arrive shortly after the previous one (median gap 29 μ s, against 62 μ s for the rest), while the daemon is still awake handling it, so they are consumed without a fresh wake-up; the 28 μ s median is the cost of that wake-up. On the long-run key-generation workload, overhead is a consistent +14–18 % across the four rounds.

Throughput and scalability. Under sequential RSA key generation, the monitor sustains 100 % detection at every load level tested (1 to 500 operations, $\sim$ 7.6 sustained ops/s), with throughput limited by key-generation cost rather than by

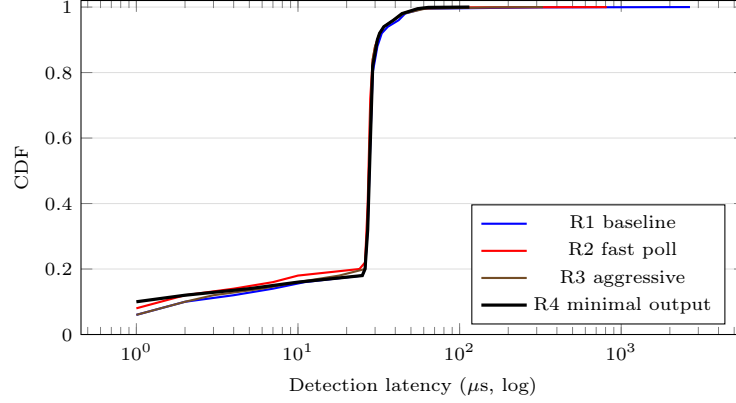


Fig. 3. Detection latency CDF by tuning round.

the monitor. With 1–16 concurrent cryptographic processes, all events are captured (48 cryptographic events at 16 processes), and median latency remains at 26–30 μs , although the P99 tail grows to $\sim 5.3 ms$ at 16 processes as the event burst saturates the consumer (Fig. 4).

Long-run stability. A continuous 3600 s run repeats the classical workload (RSA-2048, P-256, and X25519 key generation, followed by a digest and a symmetric encryption) with a 2 s pause between iterations, a lighter duty cycle than the back-to-back key generation of the throughput test. It processed 1,111,095 events (5,025 cryptographic API events, about 1.4 per second; the remainder are RNG events, most of them drawn inside the RSA key generations, since the EC and X25519 key generations draw little randomness). The event rate stayed between 292 and 338 per second in every one-minute sample. Resident memory climbs from 4 MB to a 10.5 MB ceiling over the first 52 minutes and holds there for the remainder. The climb tracks the monitor’s pre-allocated latency-sample buffer filling up and stops when the buffer does, so the growth is bounded by a design parameter. Latency over the recorded samples, the first 10^6 events that fill the buffer, was median 30 μs (P99 60 μs), within 2 μs of the median of the short latency runs in Table 4.

4.5 Real-World Applications: Static vs. Runtime

We ran nine applications drawn from the QED real-world and synthetic sets under the monitor with process filtering. Table 5 counts, for each of them, the cryptographic events observed, the randomness observations, and whether any cryptographic event was anchored by a preceding draw, together with a verdict against the expected behavior: PASS (RNG) marks randomness observed with no cryptographic operation and PASS (none) no events at all, both as expected.

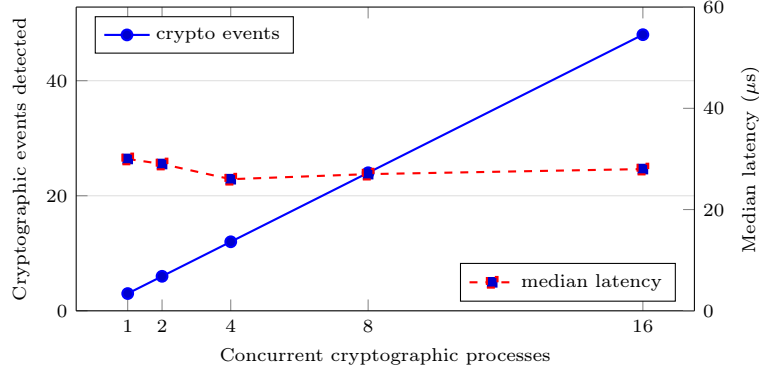


Fig. 4. Scalability: events captured and median latency vs. number of concurrent cryptographic processes.

Table 5. Runtime observation of QED applications. Static: a quantum-vulnerable library (libcrypto, libssl, libgnutls, or libtss2) among the shared-library dependencies, QED’s criterion; “-”: system binary without a recorded inventory.

Application	Filter	Static	Cryptographic	Rand	Corr	Verdict
curl (HTTPS)	curl	yes	4	9	yes	PASS
wget (HTTPS)	wget	yes	11	7	yes	PASS
ssh (version)	ssh	yes	0	1	no	PASS (RNG)
md5sum	md5sum	-	0	0	no	PASS (none)
tpm2_getrandom	tpm2_getrandom	-	0	2	no	PASS (RNG)
tpm2_createprimary	tpm2_createprim	-	4	2	yes	PASS
QED tpm2-getrandom	tpm2_getrandom	yes	0	2	no	PASS (RNG)
QED tpm2-createprimary	tpm2_createprim	yes	4	2	yes	PASS
QED synthetic RSA	openssl13.3-rsa	yes	3	2	yes	PASS

The two views agree wherever a binary that links a quantum-vulnerable library also executes one of its operations: curl, wget, the QED tpm2_createprimary build, and the synthetic RSA binary. They diverge in two cases: ssh and the QED tpm2_getrandom binary both link `libcrypto`, so a dependency inventory marks them quantum-vulnerable, yet in these runs neither executes a cryptographic operation; the runtime view records only the randomness draws. This is the presence-versus-execution distinction that motivates runtime discovery. The synthetic RSA binary is a different matter: it is not a disagreement between the two views but a runtime false negative that a probe set limited to key generation would produce, because the binary signs and verifies without ever generating a key. That is a question of probe coverage, discussed below and in Sect. 5.

The TLS workloads also expose a probe-coverage gap: key-generation probes observe the ephemeral key-exchange operations but not certificate verification,

while adding sign/verify probes exposes the verification calls as cryptographic events. At the other end, ssh’s library initialization and the TPM randomness command produce randomness observations and nothing else; both are reported as randomness observed with no cryptographic operation, and a randomness-only detector would misread them.

The synthetic RSA workload signs and verifies with a hard-coded key and never generates one. A probe set restricted to key generation therefore produces no cryptographic event, whereas adding sign/verify probes recovers three events. These events are preceded by randomness observations and are consequently marked as anchored by the monitor; the correlation establishes temporal proximity rather than proving that a particular draw was consumed by the corresponding operation. This case illustrates that runtime coverage depends on the probe manifest, a limitation discussed in Sect. 5.

5 Discussion and Conclusion

5.1 Randomness Is an Anchor, Not a Detector

Our results show that RNG events are useful as context but not as standalone evidence of cryptographic execution. As a standalone signal, randomness overapproximates: ssh initialization and TPM `getrandom` produce RNG events without cryptography. When a cryptographic API event follows a recent RNG observation, the anchor adds temporal context and raises the heuristic confidence score; it does not establish that the observed draw was consumed by that operation. An RNG burst with no subsequent known cryptographic API event can instead serve as a cue to review the probe set for an unmonitored or proprietary implementation. Such a burst is not by itself evidence of cryptography: ssh and `tpm2_getrandom` in Sect. 4.5 show the same pattern, randomness with no cryptographic call, and perform no cryptographic operation. The signal therefore marks a process for probe review rather than producing a detection, consistent with the role of D_{random} described in Sect. 3.3.

5.2 Probe Coverage

The synthetic-RSA case shows that symbol-based detection coverage depends on the probe manifest: with key-generation probes alone, the workload yields no cryptographic event, whereas adding the sign/verify paths exposes the operations observed in Sect. 4.5. The same additional probes also expose certificate-verification activity in the TLS workloads. The native ML-DSA case of Sect. 4.3 shows a similar dependency: the command-line signing path became observable only after the extended signing and verification initialization interfaces were added to the probe set. Coverage gaps are therefore an engineering matter that shrinks as probes are added. Still, the case argues for publishing probe manifests alongside CBOMs, so consumers know what a given monitor can and cannot see.

5.3 Measurement Environment

Overhead workload. The overhead in Table 4 is measured on a long-run RSA-2048 key-generation workload (500 iterations), where it holds at a consistent +14–18 % across all four rounds. A shorter workload showed run-to-run variation large enough to obscure the monitoring overhead, so the reported figures use the 500-iteration workload. A key-generation-bound workload also generates a comparatively high probe rate for this monitor because RSA key generation repeatedly reaches the monitored RNG interfaces in addition to producing the cryptographic API event itself. The long-run experiment in Sect. 4.4 records substantially more RNG events than cryptographic API events, so workloads that draw randomness less frequently are expected to trigger fewer probes.

Syscall-level anchor. The syscall-level RNG anchor needs the tracing file system that exposes syscall tracepoints. The WSL2 host did not expose it when Table 5 and the performance measurements were taken, so the `getrandom` tracepoint could not be attached and the syscall-level anchor is absent from them; for the ablation of Table 3 we ran both conditions on the same host. We additionally ran the correctness and real-world runs on a native Linux 7.0 kernel in a VirtualBox virtual machine (4 vCPUs), where the tracepoint attaches. The syscall view sharpens the ablation: every random-only negative issues `getrandom(2)`, either explicitly or through interpreter and libc start-up, so D_{random} flags every negative run (precision 0.67, F_1 0.80 in Table 3; 0.40 and 0.57 on the virtual machine, whose accuracy set had two positive workloads) while D_{api} and D_{corr} keep $F_1=1.0$; even `md5sum` shows one start-up `getrandom` event and moves from “no events” to “random only.” No application changes its cryptographic classification. Two events from unrelated processes on the shared VM were excluded using the process-name filter, while `md5sum` itself still produces no cryptographic API event. We keep the WSL2 numbers in Table 5 and the performance tables because the latency and overhead measurements come from that host; the native-kernel run is a robustness check, not a second measurement set, and is reported only here.

OpenSSL version and naming. The container’s OpenSSL 3.0 lacks native ML-KEM, so the main PQC results go through liboqs; Sect. 4.3 verifies that a single probe set also detects native ML-KEM on OpenSSL 3.5, including the algorithm name of encapsulation and decapsulation contexts. What still escapes name attribution is a key decoded from a certificate or another encoding that carries no type string, such as the peer certificate a TLS client verifies; those events are detected and anchored but reported with family unknown.

5.4 Scope

We assume a defender-side observer with root on the monitored host and dynamically linked, known libraries. Static linking requires offset-based or USDT

attachment; encrypted or obfuscated binaries are out of scope. Uprobe-based monitoring can observe sensitive call patterns, and eBPF privileges themselves are a known escalation vector in shared hosts [9]; deployments should scope collection to metadata (symbols, timings), as we do; payloads and key material are never read.

5.5 Conclusion and Future Work

In summary, we presented a randomness-anchored, eBPF-based runtime discovery system for cryptographic operations on Linux. API detection reaches $F_1=1.0$ with zero false positives on random-only workloads, and anchoring each detection on the preceding RNG events provides temporal context for each detection, with a median detection latency of $28\mu s$ and an overhead of 14–18% on a key-generation-bound workload; the static-vs-runtime case study shows the approach complements binary analysis tools such as QED, and the probe set covers sign/verify paths, ML-KEM and ML-DSA through both liboqs and native OpenSSL 3.5 paths, and live TLS 1.3 handshakes using a PQC hybrid group. Future work includes algorithm-name attribution for keys decoded from certificates and other encodings, static-binary attachment, mapping the JSON event stream to standard CBOM/SBOM schemas, and sequence-based analysis for investigating processes that exhibit RNG activity without a corresponding monitored cryptographic API event.

References

1. Caballero, J., Poosankam, P., Kreibich, C., Song, D.: Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In: Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS 2009). pp. 621–634. ACM (2009). <https://doi.org/10.1145/1653662.1653737>
2. Calvet, J., Fernandez, J.M., Marion, J.Y.: Aligot: Cryptographic function identification in obfuscated binary programs. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012). pp. 169–182. ACM (2012). <https://doi.org/10.1145/2382196.2382217>
3. Cybersecurity and Infrastructure Security Agency, National Security Agency, National Institute of Standards and Technology: Quantum-readiness: Migration to post-quantum cryptography. Joint factsheet, <https://www.cisa.gov/resources-tools/resources/quantum-readiness-migration-post-quantum-cryptography> (Aug 2023), last accessed 2026/09/14
4. Dronamraju, S.: Uprobe-tracer: Uprobe-based event tracing. The Linux Kernel documentation, <https://docs.kernel.org/trace/uprobetracer.html>, last accessed 2026/09/14
5. eBPF Docs Contributors: eBPF documentation. <https://docs.ebpf.io/>, last accessed 2026/09/14
6. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J.A., Rinetzky, N., Ryzhyk, L., Sagiv, M.: Simple and precise static analysis of untrusted Linux kernel

- extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019). pp. 1069–1084. ACM (2019). <https://doi.org/10.1145/3314221.3314590>
7. gojue: eCapture: Capturing SSL/TLS plaintext without a CA certificate using eBPF. <https://github.com/gojue/ecapture>, last accessed 2026/09/14
 8. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) Recent Advances in Intrusion Detection (RAID 2011). LNCS, vol. 6961, pp. 41–60. Springer (2011). https://doi.org/10.1007/978-3-642-23644-0_3
 9. He, Y., Guo, R., Xing, Y., Che, X., Sun, K., Liu, Z., Xu, K., Li, Q.: Cross container attacks: The bewildered eBPF on clouds. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5971–5988. USENIX Association (2023)
 10. Heninger, N., Durumeric, Z., Wustrow, E., Halderman, J.A.: Mining your Ps and Qs: Detection of widespread weak keys in network devices. In: 21st USENIX Security Symposium (USENIX Security 12). pp. 205–220. USENIX Association (2012)
 11. IBM Research: CBOMkit: A toolset for cryptography bill of materials (CBOM), and the Sonar Cryptography plugin. <https://github.com/IBM/cbomkit>, <https://github.com/IBM/sonar-cryptography>, last accessed 2026/09/14
 12. IO Visor Project: BCC: Tools for BPF-based Linux IO analysis, networking, monitoring, and more. <https://github.com/iovisor/bcc>, includes `sslsniff`. Last accessed 2026/09/14
 13. Joseph, D., Misoczki, R., Manzano, M., Tricot, J., Dominguez Pinuaga, F., Lacombe, O., Leichenauer, S., Hidary, J., Venables, P., Hansen, R.: Transitioning organizations to post-quantum cryptography. *Nature* **605**(7909), 237–243 (2022). <https://doi.org/10.1038/s41586-022-04623-2>
 14. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: An extensible approach to validating the correct usage of cryptographic APIs. *IEEE Transactions on Software Engineering* **47**(11), 2382–2400 (2021). <https://doi.org/10.1109/TSE.2019.2948910>
 15. Kwiatkowski, K., Kampanakis, P., Westerbaan, B., Stebila, D.: Post-quantum traditional (PQ/T) hybrid key agreement mechanisms for TLS 1.3. RFC 10024, Internet Engineering Task Force (Aug 2026). <https://doi.org/10.17487/RFC10024>, formerly draft-ietf-tls-ecdhe-mlkem
 16. Lacharme, P., Röck, A., Strubel, V., Videau, M.: The Linux pseudorandom number generator revisited. *Cryptology ePrint Archive*, Paper 2012/251 (2012), <https://eprint.iacr.org/2012/251>
 17. Lestringant, P., Guihéry, F., Fouque, P.A.: Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS 2015). pp. 203–214. ACM (2015). <https://doi.org/10.1145/2714576.2714639>
 18. Li, J., Lin, Z., Caballero, J., Zhang, Y., Gu, D.: K-hunt: Pinpointing insecure cryptographic keys from execution traces. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018). pp. 412–425. ACM (2018). <https://doi.org/10.1145/3243734.3243783>
 19. libbpf authors: libbpf: BPF CO-RE loader library. <https://github.com/libbpf/libbpf>, last accessed 2026/09/14
 20. Linux man-pages project: `getrandom(2)` — Linux manual page. <https://man7.org/linux/man-pages/man2/getrandom.2.html>, last accessed 2026/09/14

21. Liu, C.P.: Cryptographic asset discovery and inventory. Internet-Draft draft-liu-cadi-01, Internet Engineering Task Force, <https://www.ietf.org/archive/id/draft-liu-cadi-01.html> (Jul 2026), work in Progress. Last accessed 2026/09/27
22. Meijer, C., Moonsamy, V., Wetzels, J.: Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 555–572. USENIX Association (2021)
23. Mosca, M.: Cybersecurity in an era with quantum computers: Will we be ready? *IEEE Security & Privacy* **16**(5), 38–41 (2018). <https://doi.org/10.1109/MSP.2018.3761723>
24. Näther, C., Herzinger, D., Gazdag, S.L., Steghöfer, J.P., Daum, S., Loebenberger, D.: Migrating software systems toward post-quantum cryptography – a systematic literature review. *IEEE Access* **12**, 132107–132126 (2024). <https://doi.org/10.1109/ACCESS.2024.3450306>
25. National Institute of Standards and Technology: Module-lattice-based digital signature standard. FIPS 204 (2024). <https://doi.org/10.6028/NIST.FIPS.204>
26. National Institute of Standards and Technology: Module-lattice-based key-encapsulation mechanism standard. FIPS 203 (2024). <https://doi.org/10.6028/NIST.FIPS.203>
27. Newhouse, W., Souppaya, M., Barker, W., et al.: Migration to post-quantum cryptography – quantum readiness: Cryptographic discovery. volume B: Approach, architecture, and security characteristics of public key application discovery tools. NIST Special Publication 1800-38B (Preliminary Draft), National Institute of Standards and Technology, National Cybersecurity Center of Excellence (Dec 2023), <https://www.nccoe.nist.gov/sites/default/files/2023-12/pqc-migration-nist-sp-1800-38b-preliminary-draft.pdf>. Last accessed 2026/09/14
28. Office of Management and Budget: Migrating to post-quantum cryptography. Memorandum M-23-02, Executive Office of the President, <https://www.whitehouse.gov/wp-content/uploads/2022/11/M-23-02-M-Memo-on-Migrating-to-Post-Quantum-Cryptography.pdf> (Nov 2022), last accessed 2026/09/14
29. Ott, D., Peikert, C., et al.: Identifying research challenges in post quantum cryptography migration and cryptographic agility. Workshop report, Computing Community Consortium (2019), arXiv:1909.07353, <https://arxiv.org/abs/1909.07353>
30. OWASP CycloneDX: Cryptographic bill of materials (CBOM). <https://cyclonedx.org/capabilities/cbom/>, last accessed 2026/09/14
31. Paquin, C., Stebila, D., Tamvada, G.: Benchmarking post-quantum cryptography in TLS. In: Ding, J., Tillich, J.P. (eds.) *Post-Quantum Cryptography (PQCrypto 2020)*. LNCS, vol. 12100, pp. 72–91. Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_5
32. Park, J.W., Baek, S.Y., Jang, S.W., Han, S.Y.: Implementation methods of cryptography for preparing the quantum computing era. *Review of KIISC* **32**(6), 53–63 (2022), in Korean
33. Pixie Authors: How Pixie uses eBPF: Tracing TLS/SSL connections. <https://docs.px.dev/about-pixie/pixie-ebpf/>, last accessed 2026/09/14
34. Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.: CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019)*. pp. 2455–2472. ACM (2019). <https://doi.org/10.1145/3319535.3345659>

35. Rattanavipanon, N., Suaboot, J., Werapun, W.: A toolchain for assisting migration of software executables towards post-quantum cryptography. *IEEE Access* **13**, 4368–4380 (2025). <https://doi.org/10.1109/ACCESS.2024.3524309>
36. Shamir, A., van Someren, N.: Playing “hide and seek” with stored keys. In: Franklin, M. (ed.) *Financial Cryptography (FC 1999)*. LNCS, vol. 1648, pp. 118–124. Springer (1999). https://doi.org/10.1007/3-540-48390-X_9
37. Shin, H.S., Cho, M.J., Yu, H.S., Lee, Y.W., Tak, B.C.: A technique for accurate detection of container attacks with eBPF and AdaBoost. *Journal of the Korea Society of Computer and Information* **29**(6), 39–51 (2024). <https://doi.org/10.9708/jksoci.2024.29.06.039>, in Korean
38. Sikeridis, D., Kampanakis, P., Devetsikiotis, M.: Post-quantum authentication in TLS 1.3: A performance study. In: *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS 2020)*. Internet Society (2020). <https://doi.org/10.14722/ndss.2020.24203>
39. Sosnowski, M., Wiedner, F., Hauser, E., Steger, L., Schoinianakis, D., Galenmüller, S., Carle, G.: The performance of post-quantum TLS 1.3. In: *Companion of the 19th International Conference on Emerging Networking Experiments and Technologies (CoNEXT 2023)*. pp. 19–27. ACM (2023). <https://doi.org/10.1145/3624354.3630585>
40. Stebila, D., Mosca, M.: Post-quantum key exchange for the internet and the open quantum safe project. In: Avanzi, R., Heys, H. (eds.) *Selected Areas in Cryptography – SAC 2016*. LNCS, vol. 10532, pp. 14–37. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_2
41. The OpenSSL Project: OpenSSL 3.x EVP API documentation. <https://www.openssl.org/docs/>, last accessed 2026/09/14
42. tpm2-software project: tpm2-tss: Implementation of the TCG TPM2 software stack (TSS2), and tpm2-tools. <https://github.com/tpm2-software/tpm2-tss>, last accessed 2026/09/14
43. Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Câmara Júnior, E.P.M., Vieira, L.F.M.: Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. *ACM Computing Surveys* **53**(1), 16:1–16:36 (2020). <https://doi.org/10.1145/3371038>
44. Wang, Z., Jiang, X., Cui, W., Wang, X., Grace, M.: Reformat: Automatic reverse engineering of encrypted messages. In: Backes, M., Ning, P. (eds.) *Computer Security – ESORICS 2009*. LNCS, vol. 5789, pp. 200–215. Springer (2009). https://doi.org/10.1007/978-3-642-04444-1_13
45. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: *2017 IEEE Symposium on Security and Privacy (SP)*. pp. 921–937. IEEE (2017). <https://doi.org/10.1109/SP.2017.56>
46. Yang, Y.J., Song, K.S., Son, K.J.: Analysis of domestic and international post-quantum cryptography standardization and migration policy trends. *Review of KIISC* **34**(6), 5–11 (2024), in Korean
47. You, M.s., Kim, J.w., Shin, S.w., Park, T.j.: BPFast: An eBPF/XDP-based high-performance packet payload inspection system for cloud environments. *Journal of the Korea Institute of Information Security & Cryptology* **32**(2), 213–225 (2022). <https://doi.org/10.13089/JKIISC.2022.32.2.213>, in Korean
48. Zheng, Y., Yu, T., Yang, Y., Hu, Y., Lai, X., Williams, D., Quinn, A.: Extending applications safely and efficiently. In: *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. pp. 557–574. USENIX Association (2025), bpfime; preprint arXiv:2311.07923