Quantum Safe Explorer Differentiators

V01 (2024-04-14)

Draft

Quantum Safe Explorer analyzes application source code to discover where cryptography is being used, to detect weaknesses or vulnerabilities in this use, and to aid remediation.

This paper focuses on Explorer's differentiators vis a vis its competitors.

What Explorer Does

Before we can look at Explorer's differentiators, we need to understand what it does.

An Example

Let's start with an example in Java using the Java Cryptography Architecture (JCA). Java is, far and away, the most requested source language for Explorer and JCA is the prevalent cryptography API in Java.

This is about the simplest possible example in JCA:

```java
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.HexFormat;

public class HelloWorldCrypto {
    public static void main(String[] args) throws Exception {
        HexFormat hexFormat = HexFormat.of();
        byte[] keyBytes = hexFormat.parseHex("000102030405060708090a0b0c0d0e0f");
        SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
        Cipher cipher = Cipher.getInstance("AES/ECB/NoPadding");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] msg = "hello world     ".getBytes();
        byte[] secret = cipher.doFinal(msg);
        System.out.println(hexFormat.formatHex(secret));
    }
}
```

All this does is print encrypted "hello world" to the console. We do not hold this up as a sterling example of good use of cryptography as it involves a hardcoded key, ECB, etc. It is intended as the *shortest* possible example, and it illustrates some important points.

The action happens in *cipher.doFinal(msg)*, which encrypts “hello world”, but if you just grep for *doFinal*, as some simpler approaches might do, you don’t learn much. *doFinal()* takes a byte array and returns a byte array. However, does it take in plain text and return cipher text or take in cipher text and return plain text? *doFinal()* can do either depending on the mode that is set in the preamble call to *cipher.init()*.

If we look upward two lines above our *cipher.doFinal()* call, we can see cipher is being set to ENCRYPT_MODE so *doFinal()* is translating plain text to encrypted text. If we wanted to write our message encrypted to a file and then read it back later, we would need to reset cipher to DECRYPT_MODE.

In our example, it is fairly easy to look two lines up to find *cipher.init(),* but consider a more complex example: Suppose cipher is *init’ed* to ENCRYPT_MODE after which *doFinal()* is called repeatedly in a loop as an entire file is written block-by-block to disk in encrypted form. In that case, the *cipher.init()* might be well separated from the many *doFinal()* calls. It might even be in a different source file. For example, we might write a *writeEncrypted(data, file, cipher)* method which takes an already init’ed cipher and writes data out to a file in encrypted form.

So, now that we have found *init(),* we know which direction *doFinal()* is translating in, but we don’t know much more than that. What algorithm is being used? What key length? To find the algorithm, we need to go back still further, to *Cipher.getInstance()*. That’s where we can see the algorithm set to AES — actually AES in Electronic Code Book mode with no block padding.

And the key is set in *SecretKeySpec* but the length is implied by the *keyBytes* actual argument, which is set elsewhere.

In our very simple example, the algorithm, key, mode, etc. are all set in a half dozen lines close to each other, but this is not realistic in actual code. Consider a program which uses one algorithm and one key to write its data to disk encrypted or read it back decrypted later. The *Cipher.getInstance()* and *SecretKeySpec()* calls to instantiate the algorithm and key would almost certainly be done once during program initialization and the results stored in something like a *ProgramConstants* class. The cipher’s mode would be set to encrypt or decrypt prior to calling *writeEncrypted()* or *readEncrypted()* utility functions.

Thus, the takeaways are:

- Individual crypto calls make little sense on their own; they must be seen in context with other calls that make up standard patterns. We call this set of associated functions calls that constitute a pattern a *slice*.
- A slice may be interleaved with other, irrelevant code and may well be distributed over multiple source files. Hence, we need to apply whole program analysis. Working source file by source file is not adequate.

And remember, the example we have shown is for the simplest possible Java JCA slice. It’s easy to get a lot more complicated. We have not delved into matters like setting a JCA provider or setting *SecureRandom()*.

WHAT EXPLORER DOES

Explorer's architecture is depicted in Figure 1 below.

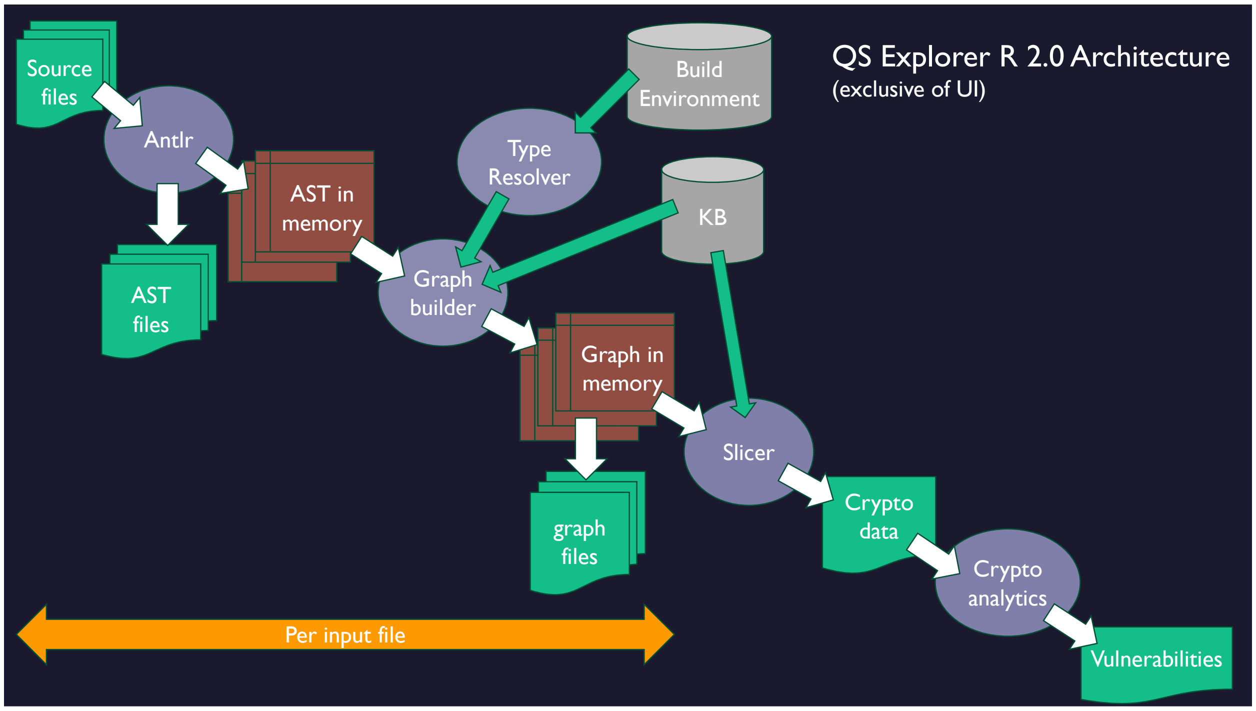
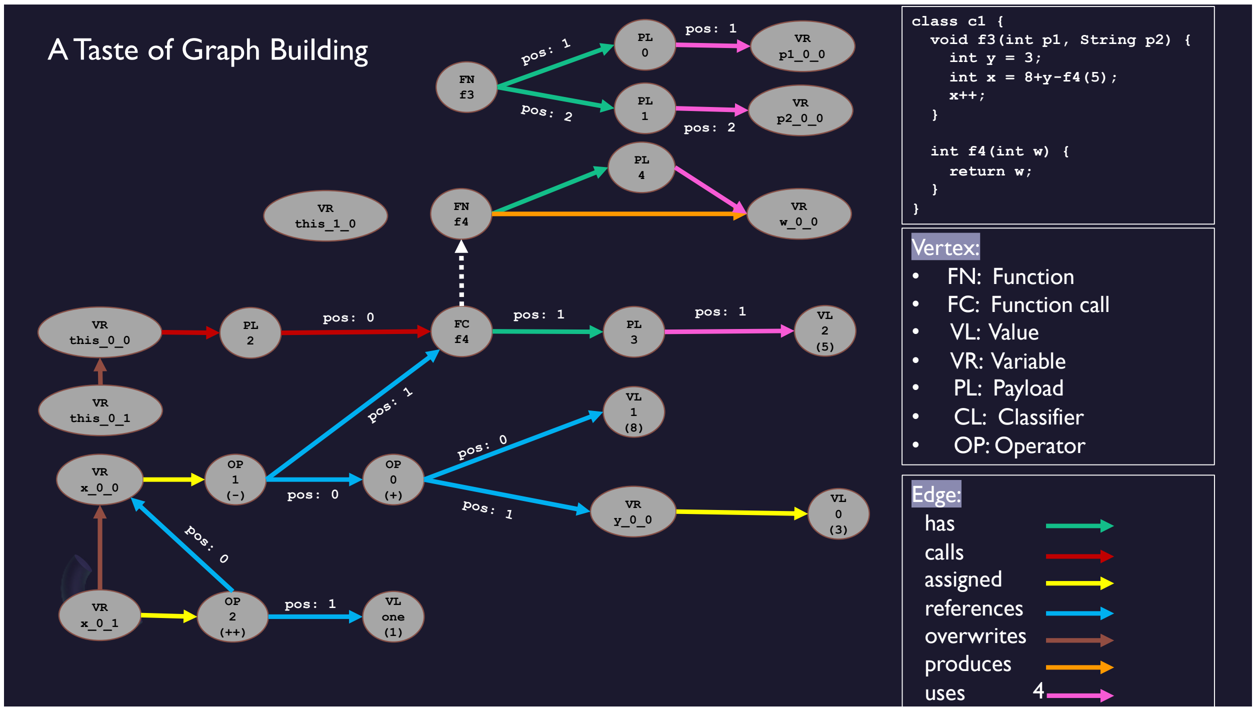


Figure 1

Taking this step by step, Explorer...

1. Scans each source file in an application and parses it to an Abstract Syntax Tree (AST).
2. Fully resolves the types of all variables and signatures of all method calls. These may be ambiguous without analyzing library imports, so Explorer analyzes imports.
3. Performs control and data flow analysis from the AST to build a graph model that tracks the state of each variable through the function.
4. Builds a call graph to track which functions call other functions. The complete graph model including control flow, data flow, and call graph is shown for a tiny example below in Figure 2.



```
class c1 {
  void f3(int p1, String p2) {
    int y = 3;
    int x = 8+y-f4(5);
    x++;
  }

  int f4(int w) {
    return w;
  }
}
```

Figure 2

5. The call graph identifies nodes that represent function calls. Explorer looks each function call up by name and signature in its Knowledge Base. The Knowledge Base enumerates for each supported language and each supported crypto library, the API of the library, including parameter signatures. The Knowledge Base also identifies key APIs which the Explorer uses as "anchor" points to begin analysis. For Java JCA, *doFinal()* is such an anchor point.

6. When a function call node maps to a Knowledge Base anchor API, Explorer uses its graph to crawl backward to find previous statements that use or set the same variables. Sometimes this leads to branching paths, as when an *if* statement, say, sets one key in its *then* block and another in its *else* block.

7. This tracing backward to establish slices transitions through the call graph so that a slice may have constituent statements in multiple source files.

8. Once slices have been computed, Explorer extracts key properties like algorithm and key length.

9. Key properties are used to identify weaknesses and vulnerabilities, like "use of a deprecated algorithm", "key length too short", "improper use of 'IVandNonce' secure random generator", etc. The actual rules are contained within the Knowledge Base, making them editable without modifying source.

We hope this is clear by now, but we shall spell it out: Explorer is fully capable of dealing with the complexities we called out in the previous section, i.e., individual crypto calls not making sense on their own, needing to be grouped into patterns or "slices" as we call them, slices transcending more than one source file.

Explorer Differentiators

Explorer is Sui Generis

At this writing, we know of no other cryptography discovery tool like Explorer, nothing that uses state of the art, whole-program, static analysis techniques to identify cryptography in context.

Source code scanning, as opposed to API hooking, network scanning, or binary scanning, is also the only technique that leads straight to code remediation. Explorer shows you exactly where your crypto is in your source code and Explorer's VS Code extension UI makes it easy to step through your crypto inventory slice-by-slice and effect remediation.

The next question would be: how easy would it be to develop a competing product? This would likely involve someone moving in from a near-adjacent space. We look at these in the following sections.

Other Cryptography Discovery Tools

There are two of note:

- SandboxAQ (formerly CryptoSense, acquired by SandboxAQ in 2022).
- InfoSec Global

SandboxAQ

SandboxAQ uses an API Hooking technique. For dynamically loaded libraries, it provides a shim layer that replaces the real crypto library. The target application thinks it is calling the crypto API, but actually gets the shim layer, which logs the call and then passes it on to the real API.

This technique has some advantages:

- You see the fully resolved actual arguments to each call.
- You see what is actually called rather than what is theoretically callable by the program.

But it has disadvantages too:

- You require a fully operational runtime environment, DEV or even PROD.
- It is very invasive. You need to install the shim layer on all servers.
- It only works for dynamically loaded libraries. It will not work on statically linked libraries.
- SandboxAQ requires a port to be opened that sends all the logged crypto data back to SandboxAQ. This is frankly unacceptable to many companies.
- It is not easy to correlate the multiple crypto calls constituting a pattern of use.

- Knowing what crypto is being called is not the same as knowing where it is being called. If you want to remediate, you need to know where the calls are in the source.

InfoSec Global

InfoSec Global has two tools. The first, AgileSec Analytics, looks for crypto use in application binaries. This is *very* hard to correlate with source code and thus does not help a great deal with remediation. Their other tool does some network analysis on endpoints exposed by applications to see what cryptography is used, typically in a TLS handshake. This requires a running environment and knowledge of which endpoint(s) the app is deployed to. Again, little direction is provided to where in the source code any problems may lie.

NETWORK SCANNERS

Nessus provides a cryptography utilization report which shows the TLS levels and cipher suites in use by each monitored endpoint. However, Nessus does not know what applications are tied to those endpoints.

CODE QUALITY TOOLS

There are several code quality tools, but we will call out two: Sonar and Code QL.

Sonar

Sonar looks for code quality problems. It does not currently identify cryptography. It does have a secrets detector that looks for secrets like keys, passwords, etc. embedded in source. This is implemented as a regular expression match, i.e. grep.

Sonar works file-by-file. It does not do whole program analytics.

We do not think Sonar's current framework is readily extensible to the sort of whole-program, slice detecting analytics Explorer performs.

Code QL

Code QL has gained traction recently partly because of Microsoft's sponsorship and partly because of its attractive architecture. CodeQL separates extraction from analysis. Its extractors analyze source and populate a database, which has schema for statements and expressions — basically an Abstract Syntax Tree represented in a database. Analyzers can work off the database.

Code QL itself mainly does file-at-a-time analysis but does a bit of whole-program analytics, mainly in the area of taint analysis. We will cover taint analysis in greater detail in the next section.

Code QL does not currently support crypto analytics.

STATIC CODE ANALYSIS TOOLS

This is another large space. Some static analysis tools are set up to do the sort of parsing, graph building, parameter tracing, slice building, and vulnerability detection that Explorer performs. That said, none of them do crypto detection today and there is reason to believe that all or most would have difficulty moving into this space rapidly.

It is important to appreciate that there is no such thing as a universal static analysis tool. Tracing every variable, every code pattern, detecting every usage would be so time and memory intensive, it would not be feasible. Instead, most static analyzers are tuned for particular usages.

The diagram below, taken from Chess & West's book "Secure Programming with Static Analysis", depicts some of the trade-offs.

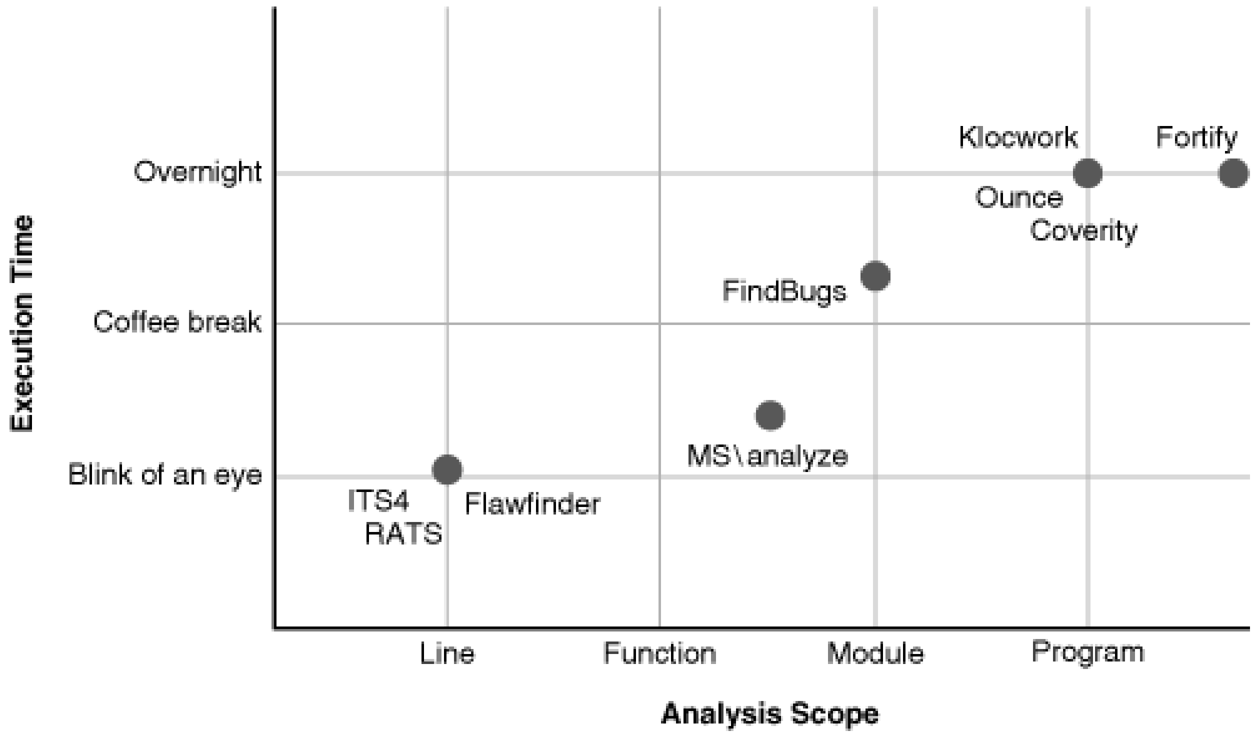


Figure 3

There is a big trade-off between speed of execution and analysis scope. Whole program analytics take time. The quicker programs, like Flawfinder, are more oriented toward code quality using line-at-a-time analysis.

These days, Fortify is probably the gold standard in whole-program static analysis looking for security vulnerabilities, but it does not detect cryptography. How easily could Fortify move into this space? The answer is not very easily.

There are two forms of data flow analysis: forward chaining and backward chaining. Fortify uses forward chaining, sometimes called taint propagation. It looks for where data enters a program — from the console, from a web client, from a file — and then propagates those values forward looking for potential buffer overrun or SQL injection vulnerabilities.

As we have seen, Explorer uses backward chaining. Explorer starts from known crypto "anchor" APIs and then traces backward to find where their relevant parameters, like algorithm and key length, are set. Backward chaining is more rare in the world of static analysis, so we think there are some moats around Explorer.

Because Explorer focuses only on cryptography and uses backward chaining, it can do aggressive graph pruning. This ought to make it much faster than other whole-program analysis tools.

Software Composition Tools

Software Composition Analysis also known as Dependency Analysis a.k.a. Software Supply Chain Analysis tools focus on finding the components used inside an application. Some work with source code, analyzing the code's imports. Others work with binaries, analyzing the signatures of known libraries. A good example is Mend.io.

When these tools work with source code at all, they don't do more than find the import statements. They do not perform sophisticated parsing and data flow analysis. Thus, they are not well positioned to do what Explorer does.

They do, however, have the potential to contribute knowledge of which third party and, perhaps, open-source libraries are used by an application. Ideally, this could be coupled with a cryptographic inventory for each constituent component to give an overall picture of the total cryptography used by an application, both directly in its own code and indirectly through libraries. The output of this analysis could be formatted as a Cryptographic Bill of Materials (CBOM).

No tool today couples software composition analysis with analysis of cryptography on a per component basis. In the future, Explorer might go there. We already analyze import statements in order to fully resolve types used in programs. It would be necessary to identify imports of open-source libraries and either scan their github repos or have a canned scan available in a database. For now, we are choosing to focus on the immediate source code of an application since we believe that is the bigger problem for our enterprise customers (see next section). We also believe that it is easier for a full static analysis tool to get into dependency analysis than for a dependency analysis tool to get into static analysis.

The Big Picture

In the foregoing, we have tried to explain what Explorer is doing and why it is difficult to replicate. But how valuable is it? In this final section, we want to broaden the context to:

- how an enterprise manages, monitors, and governs its use of cryptography,
- how it provides for crypto agility, and
- how it manages big migrations like the move to post quantum cryptography

We believe that source code analysis is important because the bulk of the applications operated by large enterprises are homegrown, i.e., ones that it writes in-house and for which it has source code. Consider the following statistics:

- JPMC
    - Operates ~7,000 apps.
    - “More than half” are developed and maintained by JPMC itself.
- IBM CIO
    - APM operates 3,860 apps.
    - Roughly two thirds are homegrown. (The 3,860 is an exact number. The “roughly two thirds” is because some of the apps in the inventory are missing the homegrown/COTS/SaaS declaration. The 2/3 is based on the fraction that has declared this and projecting it over the full portfolio.)

So, conservatively, an enterprise will have to handle any remediation of more than half its apps on its own. The remaining apps are packaged software purchased by the enterprise from third parties. While it is necessary to track the cryptography used by packaged software, this is a smaller problem since most vendors will patch vulnerabilities and handle migration to PQC in a timely manner. It is only the laggards that must be identified and chased.

So, if most of the work is to do with the enterprise’s homegrown apps, you need a tool that:

- is easy to use, non-invasive, and does not require deploying instrumentation to PROD,
- points right at the problematic code with an explanation of why it’s weak,
- can be integrated with each application’s CI/CD pipeline so that we have a continuous assessment of its cryptographic use rather than a point in time assessment,
- can roll up statistics from applications across the portfolio to give the InfoSec team a global picture of:
    - which crypto libraries are being used,
    - which algorithms, key lengths, key exchange mechanisms, etc.
    - potential vulnerabilities like hardcoded keys, etc.

While we do not exclude the value of other approaches like network scanning and binary scanning, we think whole-program source code analysis is the single best approach for cryptography. We also think that Explorer stands alone in this category and has the potential to out-distance the competition.