|  |  |
| --- | --- |
| Engineering Note | TSE-Security Tools |
| 06 | SARIF Driven Result Matching |
|  | “I guess they look the same.” |

**Abstract**

*We will discuss an algorithm for matching results across multiple SARIF logs, suitable for implementation either as a baseline processing step or as part of a results management solution.*

**Revisions**

|  |  |  |
| --- | --- | --- |
| **Date** | **Summary** | **Author/Editor** |
| 05/07/2018 | First draft | Everett Maus |

**Table of Contents**

[1. Goals 2](#_Toc513815638)

[2. Non-Goals 2](#_Toc513815639)

[3. Definitions 2](#_Toc513815640)

[4. Problem Statement 2](#_Toc513815641)

[4.1. Stressors 3](#_Toc513815642)

[5. Solution Space 4](#_Toc513815643)

[5.1. Elements of a result Identity 4](#_Toc513815644)

[5.2. Current Implementation Models 4](#_Toc513815645)

[5.2.1. In-source Suppressions 4](#_Toc513815646)

[5.2.2. Side-car Suppressions 6](#_Toc513815647)

[5.2.3. Remote Stores 6](#_Toc513815648)

[5.3. Competitor Solutions 7](#_Toc513815649)

[5.3.1. MS Internal SQL persisted results management 7](#_Toc513815650)

[5.3.2. MS Internal checked in XML results management 7](#_Toc513815651)

[5.3.3. SonarQube 8](#_Toc513815652)

[5.3.4. Semmle 8](#_Toc513815653)

[6. Guiding Principles 9](#_Toc513815654)

[7. Proposed Solution 9](#_Toc513815655)

[7.1. High Confidence Matches 10](#_Toc513815656)

[7.2. Calculating Remappings 10](#_Toc513815657)

[7.3. Lower Confidence Matching Heuristics 10](#_Toc513815658)

[7.3.1. Static Code Analysis 11](#_Toc513815659)

[7.3.2. Dynamic Analysis of UI 11](#_Toc513815660)

[7.3.3. Dynamic Analysis of a Web Endpoint 11](#_Toc513815661)

[7.3.4. Binary Analysis 11](#_Toc513815662)

[7.3.5. Fuzzing 12](#_Toc513815663)

[7.3.6. Tool Upgrades 12](#_Toc513815664)

[8. References 12](#_Toc513815665)

# Goals

We will describe an effective SARIF driven result matching algorithm, starting from normalized individual SARIF logs. We aim to perform better matching than the current state of the art, so we will also cover the current models for result matching.

We would like to articulate a substantive set of real world stressors that break baselines and describe how this algorithm overcomes them.

We will distinguish, where appropriate, between a central results store versus more distributed options, and show how this algorithm can operate in both situations.

# Non-Goals

We will not prescribe results storage or persistence mechanisms beyond covering the overall approach—that is to say, describing an efficient database storage solution for this algorithm is out of scope.

We will not cover any sort of result bucketing—that is, we won’t discuss how to group sets of results together. We see this as partly a tool problem—a tool should take results it knows are “the same” and report them together. The other half of this is a team or policy issue—a team may want to group results into a single “bug” or “work item”, which should be flexible, but falls outside of the scope of this scenario.

We will not propose or cover any sort of result state management, or handling of developer feedback on issues—that is to say, things like bug filing, result reporting, etc. are out of scope. The algorithm we provide will simply identify when two issues are “the same” versus “different”.

# Definitions

See EN05 Sarif Results Fingerprinting.

# Problem Statement

Scan tools provide a mechanism for assessing quality at a specific point-in-time. It is important to apply this assessment continuously, at some appropriate cadence, to identify new problems that emerge as a result of ongoing changes in source code, deployment environments, etc. For continuous scanning to be efficient, it is critical to correlate, where possible, individual issues (which may already have been assessed or otherwise accounted for in a results management workflow) between multiple scan runs.

To provide run-over-run correlation of scan results, results management solutions may construct ‘fingerprints’ (unique identifiers) of active results and persist them to a ‘baseline’. Two baselines can be compared by a result matching algorithm to identify which results are unique to each run and which occur in both instances. Common scenarios that argue for effective results matching:

* **‘Stop the bleeding.’** Teams with emergent maturity often wish to, as a first step, ignore all technical debt that has accumulated in code and to begin blocking new results as they are introduced. This approach has a strong appeal to developer psychology, as developers are held accountable only for quality problems that they have introduced in their current work.
* **Efficient work item creation**. Once a work item is created for addressing a static analysis result, the problem is introduced into the engineering workflow process and other signal relating to the problem should be suppressed. If results cannot be effectively tracked run-over-run, there is a risk of opening duplicate tracking items, compromising efficiency.
* **Tracking Issue Suppression**. Tool results are explicitly ignored for a wide variety of reasons, e.g., due to tool noise, identification of problems in non-shipping or soon-to-be-obsoleted code, etc. Requiring teams to revisit results that have already be determined to be out-of-scope or false positives is a significant source of team dissatisfaction.

## Stressors

The following conditions can compromise issue fingerprints, with the result that a result cannot be tracked between two versions of a code base. A result matching system will be effective according to its ability to overcome these problems:

1. **File and directory renames/moves**. Baseline fingerprints that store fully-qualified paths to files that contain quality weaknesses can break when file name and directory paths change. This class of stressor tends to break side-car suppressions; because in-source suppressions are applied to scan targets within their containing file, they are very resilient to this kind of change.
2. **Intra-file code locations**. File line and column locations associated with results will change as developers continue to modify code. Baseline fingerprints that persist this data (typically in a side-car mechanism) are therefore fragile.
3. **Logical name changes**. It can be helpful to refer to a scan target by its logical name, i.e., a fully-qualified path that identifies a scan target within a container. A .NET method, for example, can be identified by its namespace, type name, and a method identifier with sufficient details (such as return value and parameters) to uniquely identify it within the type. A fingerprint that refers to an item’s logical name can be stable even when file/directory name and location details change. It is common for developers to keep directory and file names in sync with namespace and file details, however, so the value of logical name resilience is lessened in practice.
4. **Tool updates**. Tool updates present challenges to baselining systems and require consideration (warranting its own engineering note). A new version of a tool may produce new valid results as well as false positives or introduce new false negatives (where valid results disappear). Tools may also alter log file production or other built-in suppression behavior that impact baselines. Tool updates can also result in rule identifier changes that break suppressions.
5. **Non-deterministic analysis.** Tool results may be non-deterministic, due to bugs in the tool, as a by-design feature (such as a fuzzer that randomizes its mutation of input files), or due to non-determinism in the analysis target when compiled or, for dynamic checkers, as it behaves at runtime. Web endpoint scanning is a notorious generator of non-deterministic results.
6. **Build changes**. Changes to compiler settings (or mistakenly applying a baseline produced for a ‘debug’ build to a ‘release’ scan target) can compromise baselines. UI scan targets (such as web sites) may differ due to running as a specific language, with altered user agent string, etc.). To provide a clear example of this stressor, a debug build may contain code that provokes a scan result which is compiled away in a release (because the problematic code is located within a #if DEBUG conditional).
7. **Branch merges.** Baselines are typically kept in sync with a branch. When merging a branch back into another requires rationalizing the baseline associated with each branch against a current scan run against the merged code. The practical effect is that the current scan runs need to be compared against two divergent baselines.

# Solution Space

## Elements of a result Identity

A result may be conceptually and uniquely identified by the following elements.

* **Tool**
* **Rule and rule output**. An individual result is partly defined by a conceptual notion of a quality standard (e.g., ‘do not utilize broken cryptography algorithms’). A specific result may also be denoted by the user-facing text and/or additional relevant data provided by a rule.
* **Scan target and scan target version**. Each result is associated with an entity of a specific version at a specific point in time, e.g., SomeNamespace.SomeType::SomeFunction()) as it existed in the master branch of a repository synced to a specific commit id. Scan target details can also include things like a container name, source code locations, etc.
* **Scan target creation/composition data.** An identical set of scan target files may result in substantially different outputs when compiled with differing build settings (macro definitions, optimized vs. not, etc.). It is quite common that these differences in compilation output provoke different scan tool outputs, with the result that baselines typically need to be strongly associated with a stable compiler toolchain configuration.
* **Scan data**. Every result is associated with some data that has been assessed to produce a result, e.g., a presumed code flow to a dangerous sink, or an observation that the broken MD2 hash function is used.
* **Result location**. Multiple instances of a rule must be differentiated by some logical or physical location details, e.g., several results of a rule that occur on the same line may be further qualified by column information to track them as distinct problems.

The information above is used to distill analysis results into a set of actionable results with no duplication.

## Current Implementation Models

### In-source Suppressions

In-source suppressions are indicators that are in-lined within source code to indicate that a tool result can be ignored. Scan target identifier and result locations are not rendered explicitly in an in-source suppression; instead, the in-source suppression is applied to the relevant scan target, e.g., as an attribute, annotation or code comment that is proximate to it.

#### Advantages:

* **In-source suppressions are resilient** to changes in both physical (file and directory path, file line and column data) and logical (namespace and type name, etc.) locations, a significant advantage for baselining scenarios.
* **In-source suppressions are efficient to consume.** Analysis tools are in the business of parsing target language and in-source suppressions are therefore easy to locate and consume without doing additional work (such as locating and parsing a side-car suppression file). Most tools look for and honor in-source suppressions during scan runs, without special configuration of the tool.
* **In-source suppressions provide useful documentation in some cases**. In cases where a scan tool finds a good result, that may be obvious to any developer reading the code, an in-source suppression can provide an explicit indication that a result is known but is ignored, by design. A source code comment or suppression justification may provide the rationale for this case.

#### Disadvantages:

* **In-source suppression mechanisms vary widely in form and function by source language** and, in some cases, do not exist at all. JSON, for example, does not support code comments and injecting suppressions into the JSON itself may invalidate files according to their schema.
* **In-source suppressions do not allow easy rebaselining**. It is not possible to easily rebaseline a code base that depends on in source suppressions (for example, due to updating a tool) to accept the current results of a tool as valid. Instead, any tool update requires a developer to review/fix new results and author in-source suppressions for whatever isn’t fixed. In practice, the disadvantage introduces significant downward pressure on tool servicing (which prevent tools improvements and innovation).
* **In-source suppressions clutter code.** Some subset of suppressions can provide useful documentation of why a specific issue has not been fixed. In general, suppressions are used to eliminate false positives in tools, a poor excuse for compromising code readability.
* **In-source suppressions are scan tool-specific**. Introducing a new scan tool to an existing code base can be problematic, as the source code may need to be edited in many places to create a baseline. The addition of new in-source suppressions may further clutter code that is already contains other suppressions.
* **In-source suppressions provide a single, rigid instance of review state.** In-source suppressions are most effective when used to mark an issue as permanently ignored. In practice, many scan results are temporarily ignored (e.g., when work to resolve them is scheduled for a future milestone). No existing in-source suppressions provide nuanced, machine-parsable justifications that describe why a result is suppressed.
* **In-source suppressions are not easily documented/audited**. In-source suppressions allow developers an easy mechanism to avoid satisfying compliance policy, in a way that isn’t easily audited (due to the large # of suppression mechanisms that exist). It can be difficult to produce metrics for the # of in-source suppressions in a code base for the same reason. (NOTE: To overcome this problem, SARIF producers should emit suppressed results to log files and marked them as suppressed in-source. See the result.suppressionStates information below).
* **In-source suppressions are limited in and/or provide undesirable expressiveness.** The #pragma warning disable mechanism provides an identifier and no other details for suppressed results. It is common for other suppression mechanisms to provide an open ‘Justification’ field to document the reason for a suppression, which tends to be a vector for rendering developer opinions on tool quality that aren’t appropriate to ship in product code. No in-source suppression mechanism that we’re aware of provides an expiration mechanism.
* **In-source suppressions are rarely reconsidered by development teams**. In general, there is significant pressure to suppress results to unblock the ship process. It is extremely common, even in cases where a team intends to fix a result at a future time, for results suppressed in source code never to be reconsidered.

### Side-car Suppressions

A side-car suppression is one that is persisted to a store other than files that define the entity for which a quality weakness has been found. Side-car suppressions are often stored in checked in files, with well-known names and locations. Side-car suppressions can also be persisted to arbitrary other stores, such as a remote database.

#### Advantages:

* **Side-car suppressions can be expressed in a generic format for all languages**.
* **Side-car suppressions can be more easily audited.**
* **Side-case suppressions provide clean separation of results management from production code.**
* **Multiple side-car suppression files can be used to maintain distinct views into quality.** One suppressions file could contain all current issues, for example and may be used to break builds on introducing any new problem. A second baseline might suppress all results except for a specific set that should be resolved in the current milestone. This latter baseline might be used to generate a burndown of progress towards getting clean.
* **Side-care suppressions provide open-ended expressiveness.**

#### Disadvantages:

* **Side-case suppressions are fragile when file/directory and logical name details change.** A side-car suppression must maintain a reference to a scan target location, which entails either ‘physical’ (e.g., file name + line) or ‘logical’ (namespace + type + method) details. It is very typical for expected code churn to change data that breaks these references. It is not easy to provide mechanisms that automatically update baseline references of this kind.
* **Side-car suppressions provoke merge breaks.** It is a useful practice to depend on a checked in file as a project baseline. Developers that simultaneous add new entries to the end of a list of suppressions in separate branches often break automatic merging algorithms. [One solution here is to sort suppression entries, which requires developer cooperation or additional automation]. Structured suppression files (such as JSON) exacerbate the merging problem, as these algorithms, by design, do not account for internal file formats.
* **Side-case suppressions must be explicitly referenced in order to assess result management state.**

### Remote Stores

A remote store for baselining is a system where results for a are aggregated tool run over tool run, and usually stored in some sort of database schema. They require that the scan’s results be pushed to a service and require infrastructure and the ability to comprehend different tool output formats.

#### Advantages:

* **All results are stored in a central location**. This enables central querying of tool results across runs and targets, which helps a central reporting story.
* **The store provides a single point of truth.** With sidecar files or in-code suppressions, every copy of the codebase (every branch, but also on a developer’s machine) has a copy of the baselined or suppressed issues, which may be edited or updated on that branch.
* **Depending on architecture, rebaselining can be straightforward.** The ability to simply reset the entire “state” of a codebase via a database query or similar allows easy rebaselining.

#### Disadvantages:

* **All results need to be pushed to the central location.** This has implications for scalability, but also means that endpoint authentication/authorization must be handled in some way. This makes many of these solutions not yet “cloud ready”.
* **Centralized solutions are often more difficult to move to the ‘inner loop’.** Many of these are designed to run during/after a build, and report results to developers asynchronously via bugs or emails. This is often difficult to port to an inner loop scenario.
* **Getting an idea of what results are new to a codebase requires querying the central store.** This prevents offline or disconnected runs from providing information like “what results have I added to this codebase”?

## Competitor Solutions

### MS Internal SQL persisted results management

One Microsoft internal static analysis results management system takes static analysis logs and normalizes them into a set of fixed strings for matching—consisting of the Rule Id, Scan Target’s Relative Path, and a tool-specific “Location” field (which maps in some cases to the Snippet concept in a SARIF file, but in practice is chosen per-tool). Two results match if and only if the entire set of strings are identical. Results are persisted in a per-organization SQL database, and the state is tracked file upload over file upload to enable automatic bug filing for new issues.

#### Advantages

* **Result matching algorithm is straightforward and easy to explain**. This has advantages for both tool developers (as they know how a tool change may affect result matching) and users (as a false negative match can be easily explained.
* **Result matching algorithm is reasonably successful**.This system has a large user base at Microsoft, and by and large is seen to match results accurately.

#### Disadvantages

* **This algorithm is overly simplistic.** It does not have a straightforward way to take a codeflow or stack data into account, except as some sort of flat normalized string.
* **The algorithm fails to match results.** In many cases, such as file renames, the algorithm simply fails to match the results. This leads to noise and unexpected developer alerts.
* **Mapping tool outputs to this issue model requires per tool customization.** This requires an SME to dig into the tool outputs and map them appropriately, even with shared formats (such as SARIF). This is expensive and discourages adding new tools or upgrading existing ones.

### MS Internal checked in XML results management

A second internal MS results management scheme uses a sidecar file driven baseline system, with many individual tools it runs also providing their own schemas for inline suppression. Results are grouped by ID, File, and Function (when available); then results are counted per-group.

#### Advantages

* **Straightforward**. The algorithm is easy to explain and predict the results of.
* **Robust to line changes**. If a source file is edited, as long as the results are still in the same function it will continue to track the result.

#### Disadvantages

* **Not robust to file renames.** If the relative path of a file changes, the entire set of results in the file will fail to match.
* **Not robust to logical location changes**. If you rename a function, the results will fail to appropriately match.
* **Some overmatching may occur**. If a new result is added to a group at the same time as a result is fixed, the two results will be matched.

### SonarQube

SonarQube uses first a close matching algorithm on source code content hash and relative file path, followed by a series of looser heuristics to catch file renames—first rule, line number, and line content, followed by a check for a moved block on line content, followed by looser heuristics.

#### Advantages

* **Flexible matching allows higher confidence matches to be matched first, with lower confidence matches supported by the appropriate heuristics**. This ensures that the common case is handled quickly, while still addressing possible stressors.
* **Works well with file changes and renames.** The heuristics appropriately handle situations like file renames in a reasonably flexible manner.

#### Disadvantages

* **Very source code focused.** Without line content, the algorithm cannot really match results.
* **May overmatch in some cases.** If a result is added at the same time as a result is removed, line content may be similar for where the result is detected.
* **Focus on single line content cannot reason about behavior of codeflows or similar.** SonarQube focuses on a single line, which may be insufficient to characterize many bugs.

### Semmle

Semmle uses a series of heuristics to match results across scans.

First, they match results using the locations within a file, after taking into account the file difference. They compute a file diff (using well known diffing algorithms—Myers for dense diffs with a lot of overlap, and Hunt and Szymanski for sparse diffs), which gives a set of ‘matching pairs’ and ‘difference pairs’ for each line range. For an issue on a particular location, if that location falls into a matching pair, they treat the results as matching if the locations are at the same offset from the start of the matching pair (as the code hasn’t changed). For an issue that falls into a diff pair, they look for the same issue within 3 lines of the original issue.

Then they match based on the snippet within the file—if a result is represented by the same snippet content, they treat the two issues as identical—this handles cases where the results are moved within a file, but don’t significantly change otherwise.

Lastly, they use a hash-based strategy to address file name changes—they hash the contents of 100 tokens before and after the start of the violation. If either hash matches another result, then they consider the two results to be the same. They consider matches near the start or end of a file to simply have a single hash, to avoid spurious matches on short token sequences.

#### Advantages

* **Semmle’s result matching performs well in many cases.** Their paper claims this form of result matching catches more than 99% of results, with the caveat they calculate the results commit over commit—which may be more often than necessary for compliance reasons, and has the effect of minimizing the drift between scans
* **This result matching scheme is resilient to code changes.** If a file is renamed but the result stays in approximately the same code location, the hash-based algorithm will flag the result. If a file is changed but the result remains the same, it’s tracked within the file.

#### Disadvantages

* **Heavily designed around static analysis results on code.** This algorithm would does not work for tools like binary scanners or dynamic analyzers.
* **Requires version control data or the file contents to run.** This algorithm requires full copies of every source file scanned.

# Guiding Principles

Teams should need to consider a unique result only once, so we should favor overmatching to undermatching.

We should be tool agnostic—we should be able to cover results from static and dynamic analysis tools and require little to no special tool knowledge to match results.

However, we should allow for re-baselining or other sorts of fingerprint regeneration.

The algorithm should be public and straightforward, so that tool authors can consult it and both tool authors and end users can understand when a result will be treated as identical to another result.

We will assume special knowledge of some external sources, such as tool changes and version control, but we will consume this in some neutral way (SARIF driven or otherwise).

# Proposed Solution

We propose a general algorithm for result matching between two SARIF files (presumably run on similar/the same targets). We’ll refer to one as the “baseline” SARIF file, and one as the “current” SARIF file—if a result is not matched from the baseline file to the current file it is “Absent” or “Missing”, and if a result is not matched from the current file to the baseline file, it is “New”.

The overall flow of the algorithm we are proposing consists, broadly, of three steps, which we’ll expand upon individually later:

1. Calculate deterministic matches. This step consists of matching any results we can match with high confidence—for example if a result in the current file is identical in every way to a result in the baseline, we match those two results together.
2. Calculate possible remappings. This step consists of using data in the SARIF logs, such as file hashes, file contents/diffs, etc. to calculate candidate/expected changes in Result data that we’ll need to account for while matching results.
3. Calculate lower-confidence matches in a sieve-structure. This step consists of applying a series of heuristics, from higher confidence to lower confidence, to match results. Results that are matched by a higher confidence rule are removed from lower confidence passes.

Now we’ll describe each piece of this algorithm in more detail, as well as covering approaches for cases like tool upgrades and dynamic analyses.

## High Confidence Matches

For high confidence matches—we will consider two results to be the same if every field is identical. The intuition here is that although any of the results may change run over run due to the stressors we’ve covered in [Section 4.1](#_Stressors), most results will remain the same run over run. This allows us to match the results that aren’t changing with high confidence and only consider heuristic matches for a smaller subset of the results for many common tools. This does make an assumption about both the tool’s behavior (it is reasonably deterministic in its output, for example not including the time of detection in a result), so tools that behave well will be more easily matched during this stage (reducing later work).

## Calculating Remappings

To handle situations such as file renames, we will calculate a series of potential expected changes between the baseline SARIF file and the current SARIF file. We call these “potential remappings”.

For File data, we will use a combination of heuristics—for example, if a relative file path has not changed, we assume it is unlikely to be remapped. If two files have the same hash between the baseline and the current, we will add them as a remapping. (If multiple files have the same hash, we consider either as a possible remapping.) Similarly, we note that if a file has moved to a new directory, it is possible that other files from the same directory have moved to the new directory, and we add those as candidate remappings.

If file content is embedded in the SARIF file or otherwise available, we will also consider region remappings within a single file, as well—if a file’s path is identical, we will use standard diff algorithms to compute a set of similar regions and changed regions. For matching regions, we can map a region in the file to another region in the file if it appears in the same location in the matched region. For differing regions, we will consider a region to be the same as another region if they are within the same few lines or have the same snippet content.

There may be other remappings we wish to add later on, especially in the case of a tool upgrade scenario—these are intended simply to get us started.

## Lower Confidence Matching Heuristics

For results that aren’t matched by the high confidence match algorithm, we will repeatedly apply looser heuristics in hopes of finding a match. For each heuristic, we require that the Tool and Rule ID remain the same, and we skip the heuristic if the data we require isn’t present—so for the first example, if the tool did not output codeflows, graphs, or stacks, we will skip the rule. If two results are matched by a rule, we do not consider either of them in later rules.

Some example/candidate heuristics we propose are:

1. We will consider a result to map to another result if their codeflows/graphs/stacks are the same after region/line data is removed, if any of the original containing files map to the new location(s).
2. We then consider file location (including region data) and related locations
3. We then consider file location without related locations
4. We then consider snippet and FQDN.
5. We then consider the context region.
6. We then consider tool fingerprint contributions—if all match then we will consider the results “the same”. [TODO: This section will be expanded if guidance on conventions for handling fingerprint versioning and fuzzy fingerprint matches. See: <https://github.com/oasis-tcs/sarif-spec/issues/164> ]
7. We then consider snippet without FQDN.
8. We lastly consider filename and issue counts. If a file has multiple results at this stage, we will simply map them in the order that they occurred in the file (if available), or the order they appear in the results file (if not).

If a result cannot be matched by any of these heuristics, we consider it to be a ‘new’ result if it’s only present in the ‘current’ file, and an ‘absent’ result if it’s only present in the baseline.

We expect that not all these heuristics will be required for every tool, and that it may be necessary to add new matching heuristics to address new stressors—we are simply proposing the set of these in order to provide a motivating set of example heuristics. All of these should be consistently evaluated on a per-tool basis (ideally via periodic sampling of matched/unmatched results for evaluation) for tuning and continual improvement on this section of the mapping algorithm—we expect that this part of the algorithm should be tested via sampling, and that we can do various statistical analyses on the heuristics to determine which ones are more or less effective or figure out potential candidates for addition/removal.

### Static Code Analysis

This differencing system should work well for run over run static analysis tools. Most correctly populate values such as ‘snippet’ and code regions—we can utilize those to get high confidence matches between two results. The ability to difference two files to identify how code has moved within a file also gives us high confidence that our results are accurate and valuable.

### Dynamic Analysis of UI

This is of particular interest to the Keros Desktop and Web scanning teams. We expect that in dynamic analysis scenarios, most teams will need to output specific sets of fingerprints for full or partial matches, or provide some sort of text based representation of the scope of what they’re scanning (so that differencing algorithms can go to work calculating likely changes).

### Dynamic Analysis of a Web Endpoint

Dynamic analysis of a web endpoint has several stressors—but most resolve to changes in the actual webpage or content run over run (elements are moved, etc.) or changes in the web endpoint’s behavior. If a tool provides enough data about the element when it’s output, hopefully changes in the overall location should be accurate. If a tool is reporting on specific web APIs, it should specifically omit which parts of the query string or body of the request are potentially randomly generated or may differ run over run when describing the request in the SARIF Fingerprints, and probably should remove query parameters from the actual “File” URIs that were scanned.

### Binary Analysis

In the case of binary analysis, especially if it is run on a build output, very few fingerprints are likely to remain the same run over run, aside from things like file names (or dependency hashes). However, the stressors here are also smaller—primarily build output location changes. We expect that the main stressors can probably be avoided via a combination of file name matching (as possible moves) or calculating possible folder renames (to get possible file moves from dependencies).

### Fuzzing

Fuzzing is difficult, as the results are likely to differ run over run. We recommend persisting stack values or possible code flows, as those are likely to provide the best deduplication. Actual fuzzed inputs may be provided in the SARIF log but should not make it into a fingerprint.

### Tool Upgrades

Tool upgrades provide a unique challenge, as the output format, rule names, fingerprinting algorithms, etc. may change between tool versions. We propose that a tool upgrade be handled in a particular way—you run both tools in parallel against a codebase, and then results are matched within the same run, with a slightly different set of heuristics and remappings. Keeping the targets of the tool identical between two runs is ideal, as it ensures the difference between the two tool runs will be easy to calculate. We should provide the ability for a tool owner to provide “expected changes” or “expected remappings” in case they have changed a rule name or content significantly.

# References

SonarQube Issue Matching Strategy: <https://docs.sonarqube.org/display/SONAR/Issue+Lifecycle>, retrieved May 2018

Semmle Result Matching Paper: <https://semmle.com/wp-content/uploads/2015/01/team-insight.pdf>, Published Jan 2015, retrieved May 2018