DOI: 10.15514/ISPRAS-2021-33(3)-2

Review of Static Analyzer Service Models

M.A. Menshikov, ORCID: 0000-0002-7169-7402 <info@menshikov.org>
Saint Petersburg State University,
7/9 Universitetskaya Emb., St Petersburg, 199034, Russia

Abstract. The static program analysis is gradually adopting advanced use cases, and integration with programming tools becomes more necessary than ever. However, each integration requires a different kind of functionality implemented within an analyzer. For example, continuous integration tools typically analyze projects from scratch, while doing the same for code querying is not efficient performance-wise. The code behind such use cases makes «service models», and it tends to differ significantly between them. In this paper, we analyze the models which might be used by the static analyzer to provide its services based on aspects of security, performance, long-term storage. All models are assigned to one of the groups: logical presence (where the actual computation is performed), resource acquisition, input/output, change accounting and historic data tracking. The usage recommendations, advantages and disadvantages are listed for each reviewed model. Input/output models are tested for actual network throughput. We also describe the model which might aggregate all these use cases. The model is partially evaluated within the work-in-progress static analyzer Equid, and the observations are presented.

Keywords: static analysis; integration; service model; review; classification

For citation: Menshikov M.A. Review of Static Analyzer Service Models. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 3, 2021, pp. 27-40. DOI: 10.15514/ISPRAS-2021-33(3)-2

Обзор моделей работы статических анализаторов

М.А. Меньшиков, ORCID: 0000-0002-7169-7402 <info@menshikov.org>
Санкт-Петербургский государственный университет,
Россия, 199034, Санкт-Петербург, Университетская наб., д. 7–9

Аннотация. Статический анализ программ постепенно осваивает продвинутые случаи использования, и плотная интеграция с инструментами программирования становится все более необходимой. Однако, каждая интеграция требует реализации особенной архитектуры или определенной функциональности в анализаторе. Например, инструменты для Continuous Integration обычно анализируют проекты с нуля, в то время, как тот же самый анализ с нуля малоэффективен для выполнения запросов по коду. Код, который реализует архитектуру для разных интеграций, составляет различные модели работы. В данной статье анализируются модели, которые могут использоваться статическими анализаторами, с точки зрения безопасности, производительности, долговременного хранения данных. Все модели отнесены к одной из групп на основе данных о логическом расположении вычислителя, способах получения ресурсов, методах организации ввода-вывода, а также возможностей по учету изменений и исторических данных. Описаны преимущества и недостатки моделей, приведены рекомендации по их использованию. Для моделей ввода-вывода также протестирована пропускная способность сети. Приводится модель, объединяющая все данные случаи использования. Она протестирована в разрабатываемом статическом анализаторе Equid, и в статье приведены наблюдения об особенностях её работы и реализации.

Ключевые слова: статический анализ; интеграция; модель работы; обзор; классификация

Для цитирования: Меньшиков М.А. Обзор моделей работы статических анализаторов. Труды ИСП РАН, том 33, вып. 3, 2021 г., стр. 27-40 (на английском языке). DOI: 10.15514/ISPRAS–2021–33(3)–2.

1. Introduction

Static analyzers are widely used in the industry for different purposes: defect search, verification, linting, quality assurance, code refactoring [1]. Most of these use cases can be implemented via a standard sequential model. The more projects are created the more efforts are put into developing lifelong support tools. One example is clangd [2], the tool acting as a language server [3] providing syntax highlighting, code inspections and refactoring. We believe that analysis tools have the potential to be used by a larger audience comprising not only engineers but also architects, technical management, quality assurance staff. Partially this extended audience uses analyzers nowadays, but mostly to understand code quality, while analyzers may provide more kinds of information. Currently, static analyzers are either isolated or are running locally. That limits the possibilities of the analyzer. To become agnostic to the way the analyzer is called, tools have to adopt more user scenarios and service models.

One way to approach this issue is to research how are analyzers used and in which circumstances. Combined with the technical review, classification of these *service models* would show the positive and negative aspects of each model. The paradox is that each model is so interconnected with the underlying architecture that it is hard to judge which entity is primary and which is secondary. By reviewing service models, we review the analysis architectures as well. Working out a way to support all models contributes to developing a more unified analyzer structure, improving user experience [4], and, ultimately, may lead to wider adoption of static analysis tools.

The *goal* of this paper is to classify service models that can be used by static analyzers and analyze their positive and negative networking, performance and other aspects. The *novelty* is that these models are analyzed towards application to analysis tools concerning an extended set of parameters and are combined in one model.

This paper is organized as follows. In section 2, the literature is examined. In section [3], we review all models, including logical presence models (subsection 3.1), resource acquisition (subsection 3.2), input/output (subsection 3.3), change accounting (subsection 3.4) and historic data tracking models (subsection 3.5). The most widespread models are wrapped in section 4. Then, in section 5, we define what's required for service model agnostic static analyzers. Our model-agnostic static analyzer, as well as some of the models, are tested and discussed in section 6.

2. Related work

Most works in the static analysis field explore improvements that can be applied to the analysis algorithms. The effects of service models are not typically reviewed. Common software architectural patterns [5] and patterns for data-intensive applications [6] still apply to static analyzers.

As for classification, [7] bases taxonomy on rules, technology, supported languages, configurability, etc. This separation is developer-centric, while our research is focused on the technical effects of implementation. A different approach is explored in [8], in which authors introduce a notion of *development context* comprising *local programming*, *continuous integration* and *code review* contexts. We expand further on it by exploring the service model from an analyzer's point of view, such as when handling incremental input, performing time-limited operations for IDE, etc.

The research [4] focuses on finding an answer to the question why static analysis tools are not widely adopted. One of the concerns presented by authors is that tools don't integrate into existing development processes, which intersects with our implicit thesis that industry needs more sophisticated service models. The mentioned research [8] also confirms that developers tend to avoid using the same tools for different development contexts, which means that a single analysis tool might benefit from employing more service models.

3. Models

Any software may be used via different service models. In this research, we review models based on the influence on software cooperation. Namely, the physical location influences the distance between the analysis requester and the analysis executor. In modern networking [9], such a distance is logical rather than physical since server and client might reside in the same host, so we define such models as *logical presence* models.

The second question is how are resources needed for analysis, such as input sources, headers and libraries, are retrieved. These models form a group of *resource acquisition models*.

The third problem is the propagation of input parameters from the requester to the server and the delivery of results back. This is about *input/output* models.

The fourth question is the attitude of the model to incremental analysis: *change accounting* models.

The fifth issue is similar to incremental analysis: the handling of *historic data*, such as revisions in version control systems.

In the next subsections, all these model groups are reviewed.

3.1 Logical presence models

The first theoretical model is based on where the actual computation is done. As mentioned, the location of the analysis executor is mostly logical rather than physical in presence of network namespaces (containers) and virtual machines. All reviewed models are summarized in Table 1.

Table 1. Logical presence models & their properties

Model	Security	Data leak risk	Stable connection	Network load	Environment	Performance	Score
Local computation	Low (1)	High (1)	Unneeded (3)	None (3)	Preserved (3)	Low (1)	12
Isolated computation	High (3)	Low (3)	Unneeded (3)	High (1)	Not preserved (1)	High (3)	14
Remote computation	High (3)	Medium (2)	Required (1)	Medium (2)	Manageable (2)	High (3)	13

3.1.1 Local computation

The model is widely used in static analyzer projects. In that case, the static analyzer is located on the machine requesting the analysis. The examples are LLVM and Clang [10, Svace [11], cppcheck [12] and other tools.

- *Security*: by default, the analyzer has access to all the sources and has an access to the Internet, which lowers the security in general. Moreover, access to the most data located on the host is possible. Research like [13] also stresses that the employees of companies fail to comply with security regulations. In security-critical cases, it is important to limit available file system locations by tools such as AppArmor and SELinux [14], disable internet access for the application.
- *Networking*: unused except for loopback communication or inter-process communication, which imply no use of networking hardware.
- *Performance*: developer work stations tend to have limited resources, so performance & concurrent work is limited. The solution involving the use of server-grade performant work stations is not economically effective.
- *Long-term storage*: storing artifacts for a long time is not feasible on developer work stations, except for the case when network file systems, such as NFS [15] or SSHFS [16], are involved.

3.1.2 Isolated computation

The schema is used by modern Continuous Integration (CI) tools, such as Jenkins [17], SonarQube [18], Coverity [19]. The computation is moved to a designated server that has access to committed input sources.

- *Security*: CI has all the data required to constrain allowed file system locations, for example, via SELinux, AppArmor [14]. This schema can be achieved using containerization platforms like LXC [20[, Docker [21]. Even though containers have several weak points [22, 23] and setting them up correctly requires an understanding of parameters and a modern kernel, exploiting such errors is not easy. Going forward, a designated virtual machine without direct Internet access, built solely for the static analysis of one project, is the most secure solution.
- *Networking*: such systems typically create workspaces by downloading repositories from scratch, causing significant traffic flows. However, this operation mode is usually network hardware-friendly since, as a rule of thumb, such servers have good network adapters and are connected to central switches by wire, so they are close to the repository server.
- *Performance*: the raw power needed for computations is offloaded to a server, reducing the load on developer stations to zero. Incremental operation is usually impossible due to the way workspaces are prepared and discarded.
- *Long-term storage*: storing analysis artifacts is mandatory because users might need to check results later. This shouldn't have a significant influence on disk space (since such servers have designated storage, in general) and analysis runs sporadically.

3.1.3 Remote computation with resource acquisition

The model implies that the computation is done on a separate server, but resources are acquired from developer machines via various communication channels. Clangd [2] and other language servers [3] present tools that are not technically recognizable from static analyzers but provide a similar set of services. We present the model in [24], but in this research the model is evaluated from a non-architectural perspective.

The following characteristics are seen in this model:

- *Security*: derived from isolated computation model, but data leaks are possible on the way from a local machine to a server [13]. This can be solved by using secure communication with certificate pinning.
- *Networking*: the model in which the workspace is obtained from the user directly is inefficient in the case of large projects. For example, Linux 5.10.26[1] is 1GB (174MB in tar.gz format), which would take 80 seconds (14 seconds for compressed format) on a perfect 100 Mbps link. In the case of compressed format, it takes 6 seconds to unpack on Intel Core i7-7700HQ based laptop with Samsung 980 Pro SSD, Ubuntu 20.04. Compressing to this format takes 30 seconds on the same host. That means that, if the workspace is obtained from the user, the complete transmission time is 80 seconds or 30 + 14 = 44 seconds (considering the receiver a more advanced host with higher unpack performance). The link is, however, usually not perfect: for example, WiFi links are ailing from network congestion [25], decreasing available bandwidth even further. The viable option is collecting changes from the revision known to the static analysis host (this option is discussed in subsection 3.4).
- *Performance*: the computation is offloaded to the high performant server, with no load to developer stations. The incremental operation is possible in case snapshots of the internal state are stored by the static analysis host.
- *Long-term storage*: storing analysis artifacts is also mandatory, the influence on sparse runs is the same as for isolated computation, however, significant disk space might be consumed by

[1] https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.10.26.tar.gz

per-developer incremental runs. As a result, the recommendation is to prepare a mechanism for discarding old per-developer analysis results.

3.2 Resource acquisition models

All resource acquisition models are reviewed in Table 2.

Table 2. Resource acquisition models & their properties

Model	Input length	R&D efforts	Preparatory work	Stable connection	Compiler compatibility	Score
Local resources	Optimal (3)	Low (3)	None (3)	Unneeded (3)	Full (3)	15
Shared repository	Moderate (2)	Low (3)	None (3)	Unneeded (3)	Full (3)	14
Preprocessing	Large (1)	Low (3)	High (1)	Unneeded (3)	Absent (1)	9
Pre-tracing	Moderate (2)	Moderate (2)	High (1)	Unneeded (3)	Full (3)	11
Virtual file system	Optimal (3)	High (1)	Low (3)	Required (1)	Full (3)	11

3.2.1 Local resources. Shared repository

These two models just define the typical schemas used in software engineering. The local resource model is used in all tools running locally, such as compilers, static analyzers. The shared repository model is enforced by continuous integration environments.

3.2.2 Preprocessing

In this schema, the input is preprocessed locally and the analyzer gets a preprocessed version for further analysis.

- *Input size*: preprocessed definitions are very large. The file with only \path{<iostream>} header included and an empty main() is 49 bytes long, while the preprocessed version is 751954 bytes long (GCC 9.3.0 on Ubuntu 20.04).
- *Analysis*: the problem might be the preprocessor's output is not compatible between the requester and analyzer hosts. This is better seen if source and target hosts have different operating systems and toolchains.

If it is clear that the compiler used on both localhost and analysis host matches or at least is compatible, and the analysis runs on one input file at most, then this schema might be a simple and cost-effective solution for the implementation of remote analysis.

3.2.3 Pre-tracing of dependencies

The core idea is to perform tracing of all needed files before sending an analysis request. This process can be not straightforward. Tools like Build EAR [26] intercept commands passed to compilers, but don't provide lists of all needed files. This tool can be used in conjunction with utilities tracing system calls to get this information (such as strace[2]).

- *Input size*: reasonable since it includes only needed files.
- *Analysis*: requires integration of virtual file system with pre-downloaded files into parsing stage.

This model is similar to preprocessing, however, files are packed into request individually. This schema is less problematic than preprocessing because files are not present in the request twice or more times, reducing the cost of networking transfer.

[2] https://github.com/strace/strace

3.2.4 Virtual network file system

A virtual network file system is a technique that can be used to acquire resources from a source host on-demand. It can be used through the well-known implementation such as NFS [15] and SSHFS [16], or via a custom protocol. This schema has the following properties:

- *Input size*: optimal because taken on demand (in case files are cached on a server).
- *Analysis*: requires integration with the parsing stage. This model reduces the cost of analysis in case of early termination which may occur if an input has obvious syntax defects.
- *Networking*: needs a stable connection between a local host and an analysis host. It can be problematic considering that a significant part of hosts is behind Network Address Translation (NAT) [9] gateways and thus doesn't have a fixed IP. In such systems, the hosts need to use keep-alive techniques to avoid early preemption of entries in gateway NAT tables. Also, the use of well-known implementation may exhibit the problem of passing traffic through in case the static analysis client is behind NAT or a firewall and the implementation uses the pipe in the direction from server to client.

3.3 Input/output models

All input/output models are reviewed in Table 3.

Table 3. Input/output models & their properties

Model	R&D efforts	Stable connection	Network load	Notifications	Score
CLI	Trivial (3)	Unneeded (3)	None (3)	Unneeded (3)	12
Stateless client/server	Trivial (3)	Unneeded (3)	High (1)	Impossible (1)	8
Stateful client/server	Moderate (2)	Required (1)	Low (3)	Possible (3)	9
Streaming model	High (1)	Required (1)	Low (3)	Possible (3)	7

3.3.1 Command line interface model

This model is widespread in the industry. The input is provided with input arguments and input stream, the output – with the result code and stdout/stderr stream.

3.3.2 A stateless client/server model

The input is the request to the server, the output is a response to the request.

- *Networking*: this model implies that after the request is sent, the response must follow after analysis is done, not necessarily to the same request (might be a status request).

  The problem with this model is that notifications need a side-by-side implementation (i.e. a communication channel directed towards the client). Without notifications, the status polling is redundant, but not harmful due to small absolute packet sizes.

  A significant performance issue in real conditions may occur if a large amount of input data is sent over short-living TCP sessions. The reason is that most home-grade gateways accelerate network traffic only if the session reaches a specific number of packets (e.g., 5). Shorter sessions may appear unaccelerated and may be processed via CPU, not reaching a maximum practically performance (in the author's experiments with 1gbit links, the accelerated performance tops at 940 Mbps, while unaccelerated traffic reaches 50 Mbps, at most).
- *Practical aspects*: the approach can be implemented within the REST paradigm, which has many available implementations for any platform.

Practically, this limits the usage of the model to short requests. That's the reason the model is used within Continuous Integration systems, data management cases (such as the configuration of services like Jenkins, GitLab; manipulation of objects in bug trackers, etc). In other cases (e.g. compiler support case), this paradigm is not efficient.

3.3.3 A stateful client/server model with or without notifications

The input is a series of requests to the server, the output is a series of responses from the server.

- *Networking*: this model is efficient regarding networking hardware in the case of long-term TCP sessions. Most traffic will be accelerated, so the maximum performance will be demonstrated.
- *Practical aspects*: the model requires a custom state machine, notification system. The development cost is higher.

3.3.4 A streaming model

This is a variation of the client/server model, so the throughput is nearly the same. The input seen on the server is dynamically formed by requests, the computation is performed for currently known data.

3.4 Change accounting models

3.4.1 Fixed revision

The analyzer pulls the specific version of a source. If it is needed to re-analyze some part of the code, the complete analysis is performed.

- *Time*: complete execution every time.
- *Analysis*: requires no special handling from the analyzer's side.
- *Networking*: download of the complete repository might take significant time, however, this process is unconditionally networking hardware-friendly.

This schema is suitable for Continuous Integration processes, but long analysis time blocks the interactive use cases.

3.4.2 Incremental updates

The analyzer builds the model of a program on the first run. If the user decides to reanalyze a file or two, changes are obtained incrementally.

- *Time*: slow once, fast on incremental updates. However, in the case of global changes, the analysis time might increase dramatically, reaching the complete time or even overcoming it due to preliminary dependency graph analysis.
- *Analysis*: puts additional requirements, such as discardable state that is trivial to invalidate when a part of dependency graph changes. Dependent parts of the state should be rewritable.
- *Networking*: the difference between projects typically has negligible size compared to complete repository, so the process of obtaining differences is networking hardware-friendly, especially in the case of one TCP session or the same UDP source/destination addresses and ports.

3.4.3 Daily updated global revision with incremental user-defined changes

A typical use case would be that the analyzer runs every night on the latest revision, but if the user requests the analysis of a diverging source, the «latest» revision is forked and only differences are reanalyzed.

- *Time*: this schema improves analysis performance for developers running the analysis on a large codebase with minor differences.

- *Analysis*: the incremental schema requirements plus scheduling of daily updates, temporary storage of analysis artifacts.
- *Networking*: developers typically don't change the large codebase significantly. Because of that, the difference is ought to be minor, and the network load is the same as for the incremental schema.

If the analysis state is transferrable, the developers might cache the state and run the analysis locally. This is possible for some analysis kinds, such as code queries, dependency analysis.

3.5 Historic data tracking models

Some analyzers might take advantage of historic data. In addition to the usual code metrics changing over time, the practically useful case would be to narrow down a revision with a specific defect not tracked by analyzers (i.e. logical mistake)

3.5.1 A model without tracking of historic data

- *Analysis*: trivial to implement compared to a model with tracking.
- *Data storage*: only needs one specific revision, no extra data is needed.

3.5.2 A model with complete snapshots of historic analysis data

- *Analysis*: requires meta run of analysis over two or more revisions, which complicates the structure of analysis.
- *Data storage*: the analysis data for all revisions in question should be collected.

3.5.3 A model with differential snapshots of historic data

- *Analysis*: more complicated compared to the model with complete snapshots, additional invalidation of data is needed. That also requires maintenance of algorithms for propagating analysis data differences, which may make the complete task difficult.
- *Data storage*: analysis can be done once, and then only analysis database differences can be stored.

Model without tracking is trivial to implement. Models with snapshots may support use cases in which historic data is important, but it comes with a cost of extra time, data storage (high in the model with complete snapshots) and development complexity (high in the model with differential snapshots).

4. Combination shortcuts

After review of basic models, it is obvious that their combinations are already used worldwide:

1) **Local (incremental) model** – local computation, local resources, command line or server model with a fixed revision (incremental updates) and no tracking of historic data.
2) **Continuous Integration model** – remote computation, source repository, stateless client/server model with custom notifications, fixed revision, no tracking of historic data.

5. Considerations for service model agnostic static analyzers

Considering suggested use cases, it is possible to form suggestions on what should be done in a static analyzer to support more these models (fig. 1)

Logical presence models and input/output models are tightly coupled. A service model agnostic analyzer should have an abstraction layer for the complete execution – the job subsystem.

Resource acquisition methods imply that there must a separate abstraction layer for retrieving file data from different hosts.

Incremental change support implies that objects must be addressable in a unified and interchangeable manner, so that older object versions might be discarded, while new versions added as-is. This should be done right after retrieving data and remote resources.

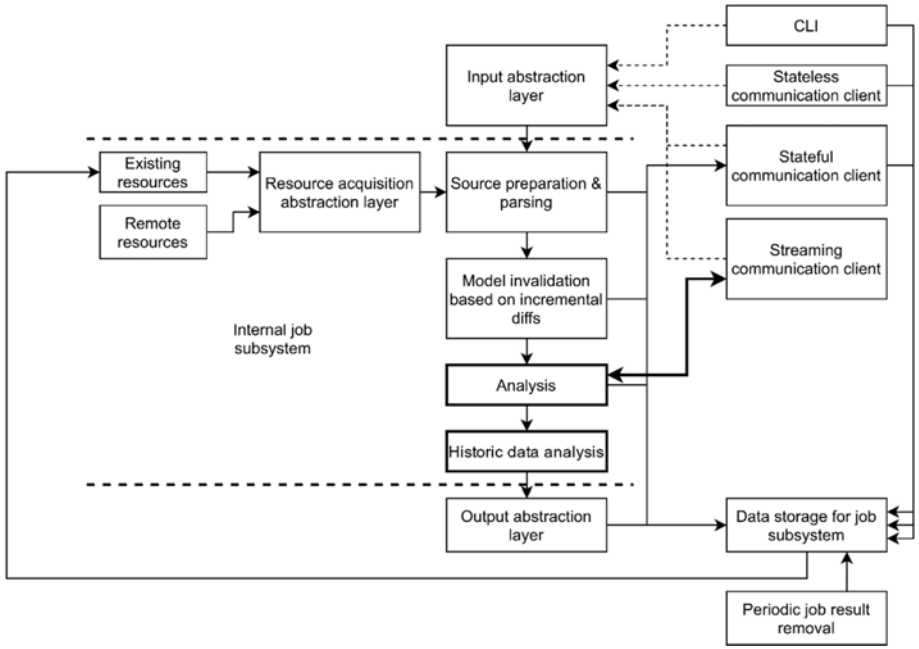

Fig. 1. Possible schema for model agnostic static analyzer

To facilitate status polling, incremental change handling and historic data tracking, the output should be saved to data storage, accessible for extended periods. Historic data tracking also implies having a subsystem of meta-analysis, which allows reviewing deltas between revisions.

6. Testing and discussion

6.1 Characteristic-based evaluation

The characteristic-based evaluation of models was performed in tables 1, 2 and 3. For each characteristic, a numeric value ranging from 1 (worst) to 3 (best) had been chosen. The total score for each model is written in the column «Score». This evaluation is partially subjective but had been discussed with a few experts in relevant domains.

The results are as follows. The best model among logical presence models is an isolated computation, which is confirmed by its popularity in the software engineering industry. The second model is a remote computation with resource acquisition. It combines the high performance of isolated computation with manageable customization to comply with the environment and use cases. The third model is a local computation. The problems of this model lie in practically low performance and high data leak risk (the developer machine is likely to be insecure). However, if this risk is diminished by using a secure operating system and working firewall rules, this model would share the score with remote computation.

Among resource acquisition models, classical local and shared repository models are the best. When considering models for non-classical use cases, pre-tracing of dependencies and virtual file system are better choices than preprocessing method. But, in practice, preparatory work for tracing might

be time-consuming, making the provision of thin clients hard. So, in the author's opinion, the virtual file system is the preferred choice for non-classical use cases.

For input/output models, the choice is related to the use case even more than for previous models. However, when choosing among networking models, stateful client/server communication is preferred as it reduces network load, provides notifications (reducing polling) while keeping R\&D efforts moderate.

The preference between change accounting models is unambiguous. The incremental model supports more use cases, and at the same time, the daily updated revision enhances it with much better performance in common developer routines.

Historic data tracking stays a little apart from this comparison. The more data is processed, the more time is taken and the more useful data is carved, therefore it is hard to name the preferred model.

6.2 Evaluation in static analyzer project

The considerations for service model agnostic analyzers were used as a basis for our project – Equid static analyzer [27]. We emphasize that the project is not following the schema in all ways since there is a lag between design and actual implementation. Our implementation includes a frontend library – the part that manages jobs for a specific workspace. The frontend library is used by the command line interface and server binary, both of them construct the workspace and fill it with job types, paths and environment information. The job types define the semantic visitors that are invoked at the end of analysis stages, during meta run, and have an impact on the selection of verification rules. The frontend library starts the analysis and provides an interface to get the current status or stop execution if needed. After finishing all jobs related to a specific run, the user might obtain the result of the analysis in all requested forms. The supported forms are defects, dependency analysis, call graphs, language identification.

The incremental analysis model lags behind the design. The support of incremental analysis is built into an object database, and it is possible to discard old objects and then drop new objects in. There is a saved dependency graph that can be used to invalidate parts of the analysis run. However, the incremental analysis support is not finished yet and we can only experiment with it. In our testing, if the incrementally changed file makes 10\% of input size, then the time to recompute it will match 20.07% (on average) of time taken for the whole input due to the need to invalidate the map. In case of excessive dependencies between updated and untouched files, the computation might take up to 40% of the original analysis time, although it is possible to design a case that will invalidate the complete program model.

The supported mediums are JSONRPC[3] and binary streaming over TCP with TLS enabled. These mechanisms are implemented in a straightforward manner and are adequate considering networking and security requirements.

During the evaluation, we have found that the optimal model effectively falls back to trivial software architecture if some functionality is not needed. When they are needed, extra stages get enabled and start adding expected diagnostic data to reports. That is the reason why it is possible to experiment with unfinished functionality in Equid's architecture. This is an advantage of the model.

The other advantage is a clear decomposition between *the core* and *the service*. The analysis functionality is a black box for the service. The service part provides input arguments, takes notifications provided by the core, passes streaming data to the analyzer and reuses the output as many times as needed, however, those are only extension points available. As seen in the schema, the main part remains sequential, therefore, still simple for development.

There are certain problems. While the *simple* design matches the *complex* architecture, imminent conditional jumps still make performance penalties. Also, it is harder to maintain the support of these service models, though this issue may be neglected by keeping the core as minimal as possible.

[3] https://www.jsonrpc.org

6.3 Network performance

As for network performance numbers, we performed testing of:

- Stateless polling versus notification model. In the case of using exponential backoff variation (5, 10, 20, 40, 80, 160 seconds at most), there are around 294 bytes per request and 210 bytes per response (Table 4). In the case of notifications (Table 5), a response is around 140 bytes and keepalive packets are around 70 bytes (Analysis start/destroy is not considered for the case of polling, TCP session instantiation/finalization is not considered for notifications). The time difference is large between polling and keepalive models, but in absolute numbers, these differences don't impact allocated bandwidth significantly and thus might be ignored.

Table 4. Data transfers with polling

Total time (sec)	Start (sec)	End (sec)	Steps	Delta (sec)	Data transmitted (bytes)	Data received (bytes)
630	5	160	8	5	2352	1680
95	5	160	5	60	1460	1046
13080	5	160	86	35	25542	18232

Table 5. Continuous data transfers

Total time (sec)	Keepalive packets (pkts)	Data transmitted (bytes)	Data received (bytes)
634	10	700	140
92	1	70	140
13189	219	15330	140

- Data transfers over WiFi (Table 6). A dual-band home gateway based on MediaTek platform with IEEE802.11n and IEEE802.11ac bands was used for testing. The test server is connected to the gateway over the 2.4GHz band (actual frequency is 2.412GHz), the client is connected to the gateway over the 5GHz band (actual frequency is 5.3GHz). For single-thread TCP performance, the data has been sent in the biggest possible packets according to MTU/MRU in the network. For SSHFS, the data has been sent file by file. The actual performance numbers demonstrate that the preprocessing schema is, indeed, slower due to higher input size. The difference between single-thread TCP with raw input is around 18.57 Mbps (21.6% of raw TCP performance), however, this difference may be either judged by the simpler implementation of SSHFS. On the other hand, a possible reduction of input based on the existence of files on the server not only in one user's sandbox might have a positive impact on the performance of custom protocols based on TCP. At the same time, the local model has zero penalties on file transfers and this result cannot be surpassed.

Table 6. Source code transfers

Approach	Total time (sec)	Input length (MB)	Links	Throughput (Mbps)
Single-thread TCP (raw input)	9.85	101	5.3GHz → Gateway → 2.412GHz	82
Single-thread TCP (preprocessed input)	48.75	470.66	5.3GHz → Gateway → 2.412GHz	77.2
SSHFS (raw input)	12.054	101	5.3GHz → Gateway → 2.412GHz	67.03

6.4 Limitations of the approaches and further development

The proposed schema of the service model agnostic analyzer aggregates models in a straightforward manner. The problem with it is that it is not optimized as there was no research on the most optimal

model. In our view, an improvement can be achieved if some numerical quality measure for service model combinations is proposed.

The problem with the comparison of models is that it is biased towards implementation. The most widespread cases were carefully chosen, such as source code transfer evaluation or polling versus continuous data transfer testing, however, actual implementation may work around negative aspects shown in the paper. That may happen since analyzers cannot be seen as pure implementations of these discrete models. Combining models for reaching the best quality of output model is encouraged, even if complete aggregation is not in question.

Also, as the research's goal is to study common models and their generalizations, it is often the case that a widespread example of the specific model does not exist, and we have no resources to implement all of them in the analyzer with sufficient detail level. That limits model reviewing possibilities. A further improvement would be achieved after developing such examples (toy analyzers) and verifying them on many samples.

6.5 Suggested use cases

These models may work on different occasions. Based on the review of models, we propose the following mapping from use cases:

- **Complete project and inter-project analysis**: based on the advantages of isolation, the continuous integration model seems a better choice.
- **Basic reference search, refactoring**: since these use cases don't imply deep project inspection [28], a local (incremental) model should be optimal.
- **Code queries** [29]. Depends on the size of a project: small projects might be analyzed locally in a separate instance of the analyzer. Big projects with a distributed team mostly sharing the same source may take advantage of remote computation with a virtual file system, a stateful client/server model, a daily updated global revision with incremental changes model and historic data tracking.
- **Project import & dependency analysis**. Depending on the requirements such as the location of the project and its size, the preferred model might range from a simple *local model* to a remote computation (with or without a virtual file system), source repository and a fixed revision model.
- **Debugger support** – analyzer supports debugger with code insights (e.g., similar model is seen in [29]). The local model is sufficient for small projects, but large projects should be analyzed within the remote computation, virtual file system, daily updated global revision and incremental updates model.
- **Compiler supporting model**. In that case, the compiler does code generation, but the analyzer supports it with additional inferred contract checks, the information about clearly unsatisfied assertions, et cetera. Local computation, local resources, streaming model, fixed revision.
- Static/dynamic analysis cooperation. Such cooperation is suggested by FSTEC [30] «Protection against unauthorized access to information» certification. For example, a dynamic analyzer might trace the execution to let the static analyzer verify that all traces are valid. It might be done in a remote execution model with a virtual file system, daily updated global revision with incremental updates.
- **Technical documentation preparation**. Also a part of FSTEC [30]. Usually, the process is done once at the end of a release cycle. Considering the importance of precision, Continuous Integration is the most efficient model.

6. Conclusion

The service models that can be used by static analyzers were described. This list includes logical presence, resource acquisition, input/output, change accounting and historic data handling models. An aggregate model enabling significantly diverging use cases is presented. It was tested in a real-

world static analyzer and demonstrated technical advantages and disadvantages. Part of the models was compared directly by characteristics, and recommendations for model selection were provided.

Список литературы / References

[1] D. Binkley. Source code analysis: A road map. In Proc. of the Symposium on Future of Software Engineering (FOSE '07), 2007, pp. 104-119.

[2] What is clangd? Available at https://clangd:llvm:org, accessed 15.03.2021.

[3] Langserver.org - A community-driven source of knowledge for Language Server Protocol implementations. Available at: https://langserver:org, accessed 15.03.2021.

[4] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In Proc. of the 2013 International Conference on Software Engineering, , 2013, p. 672-681.

[5] M. Richards. Software architecture patterns. O'Reilly Media, 2015, 47 p.

[6] M. Kleppmann. Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems. O'Reilly Media, 2017, 616 p.

[7] J. Novak, A. Krajnc et al. Taxonomy of static code analysis tools. In Proc. of the 33rd International Convention MIPRO, 2010, pp. 418-422.

[8] C. Vassallo, S. Panichella et al. How developers engage with static analysis tools in different contexts. Empirical Software Engineering, vol. 25, no. 2, 2020, pp. 1419-1457.

[9] A.S Tanenbaum and D.J Wetherall. Computer networks. Pearson, 5th edition, 2010, 960 p.

[10] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proc. of the International Symposium on Code Generation and Optimization, 2004, pp. 75-86.

[11] В.П. Иванников, А.А. Белеванцев и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014 г, стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7 / V.P. Ivannikov, A.A. Belevantsev et al. Static analyzer Svace for finding defects in a source program code. Programming and Computer Software, vol. 40, no. 5, 2014, pp. 265-275.

[12] Cppcheck - a tool for static C/C++ code analysis. Available at http://cppcheck:sourceforge:net, accessed 15.03.2021.

[13] F. Bélanger, S. Collignon at al. Determinants of early conformance with information security policies. Information & Management, vol. 54, no. 7, 2017, pp. 887-901.

[14] Z.C. Schreuders, T. McGill, and C. Payne. Empowering end users to confine their own applications: The results of a usability study comparing SELinux, AppArmor, and FBAC-LSM. ACM Transactions on Information and System Security, vol. 14, no. 2, 2011, pp. 1-28.

[15] S. Shepler, B. Callaghan et al. Rfc3530: Network file system (nfs) version 4 protocol, 2003. Available at https://www.rfc-editor.org/info/rfc3530, accessed 15.03.2021.

[16] M.E. Hoskins. SSHFS: super easy file access over SSH. Linux Journal, no. 146, 2006, pp. 1-4.

[17] J.F. Smart. Jenkins: The Definitive Guide: Continuous Integration for the Masses. O'Reilly Media, 2011, 404 p.

[18] G.A. Campbell and P.P. Papapetrou. SonarQube in action. Manning Publications, 2013, 392 p.

[19] A. Bessey, K. Block et al. A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM, vol. 53, no. 2, 2010, pp. 66–75.

[20] K. Ivanov. Containerization with LXC. Packt Publishing, 2017, 352 p.

[21] D. Merkel. Docker: lightweight Linux containers for consistent development and deployment. Linux journal, no. 239, 2014, pp. 1-2.

[22] J. Wenhao and L. Zheng. Vulnerability analysis and security research of Docker container. In Proc. of the IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE), 2020, pp. 354-357.

[23] T. Combe, A. Martin, and R. Di Pietro. To Docker or not to Docker: A security perspective. IEEE Cloud Computing, vol. 3, no. 5, pp. 54-62, 2016.

[24] M. Menshikov. Towards a resident static analysis. Lecture Notes in Computer Science, vol. 11620, 2019, pp. 62-71.

[25] Z. Hays, G. Richter et al. Alleviating airport WiFi congestion: An comparison of 2.4 ghz and 5 ghz wifi usage and capabilities. In Proc. of the Texas Symposium on Wireless and Microwave Circuits and Systems, 2014, pp. 1—4.

[26] rizsotto/bear: Bear is a tool that generates a compilation database for clang tooling. Available at https://github:com/rizsotto/Bear, accessed 15.03.2021.

[27] M. Menshikov. Equid – a static analysis framework for industrial applications. Lecture Notes in Computer Science, vol. 11620, 2019, pp. 677-692.

[28] M. Fowler. Refactoring: improving the design of existing code. Addison-Wesley Professional, 2nd edition, 2018, 448 p.

[29] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. in Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005, pp. 365-383.

[30] Federal Service for Technical and Export Control. Available at https://fstec:ru, accessed 15.03.2021.

Информация об авторе / Information about the author

Максим Александрович МЕНЬШИКОВ, аспирант кафедры системного программирования. Научные интересы: статический анализ, обратная разработка, инструменты разработки, высокопроизводительные вычисления, виртуализация.

Maxim Aleksandrovich MENSHIKOV, PhD student. Research interests: static analysis, reverse engineering, development tools, high-performance computing, virtualization.