UNIVERSITÀ DEGLI STUDI DI UDINE

SERICS

Italiadomani

Finanziato dall'Unione europea
NextGenerationEU

Ministero dell'Università e della Ricerca

DBCChecker: Formally Verifying Security Properties of Containers Compositions

based on joint work with A. Altarui (U. Milano), M. Paier (IMT Lucca), and others

5G and Cloud Security Awareness Days, Rome, 20/06/2024

Marino Miculan, Univ. Udine & Venezia
marino.miculan@uniud.it

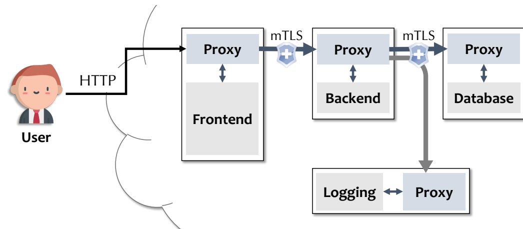
Containers are widely adopted in the Cloud

Containers are increasingly adopted in the design and implementation of complex software systems, especially in the Cloud.

Containers are lighter, more efficient alternative to Virtual Machines

Support Microservice-oriented architectures: fine granularity services and components

Simplify horizontal and vertical scalability

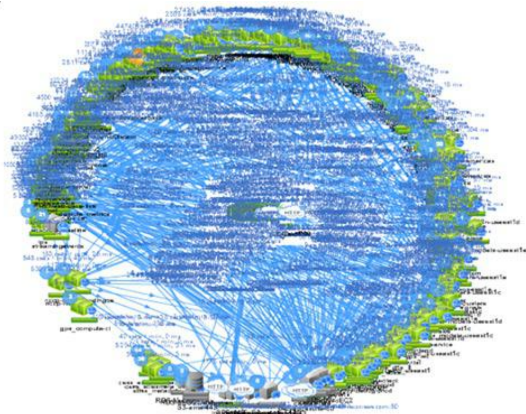


Containers enforce weaker separation than VMs

Applications can be composed by hundreds or thousands of containers.

A cloud provider often runs many applications (possibly from different clients) on the same infrastructure

Connecting and coordinating containers into a complete working system is not trivial.



Containers enforce weaker separation than VMs

Applications can be composed by hundreds or thousands of containers.

A cloud provider often runs many applications (possibly from different clients) on the same infrastructure

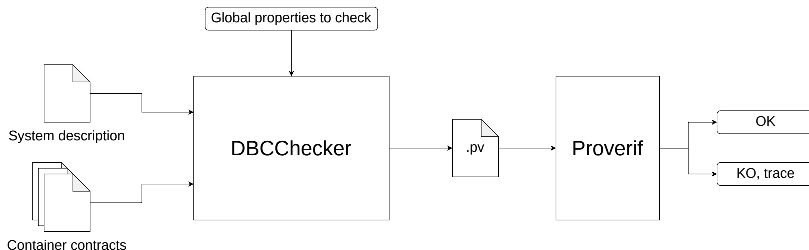
Connecting and coordinating containers into a complete working system is not trivial.

Violating security goals and policies through misconfiguration is easy.

DBCChecker

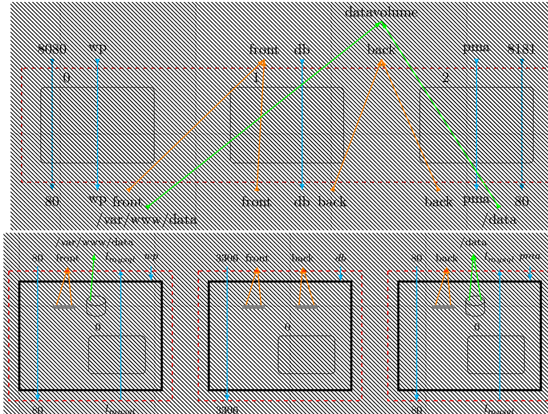
We propose **DBCChecker**, a tool that aims to verify security properties of systems obtained by the composition of containers. It needs:

1. a configuration of a container-based system
2. for each container an abstract description of the interaction on its interface



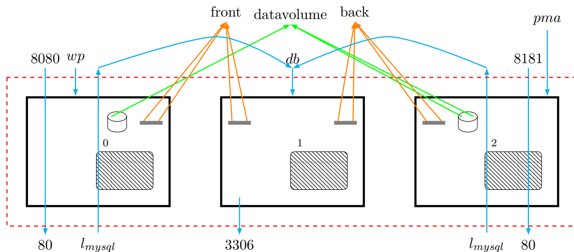
Bigraphical models of containers

DBCChecker is based on the abstract bigraph based model of container based systems presented in (Burco, Miculan, Peressotti, ACM SAC 2020), and the JLibBig implementation of bigraphs.



Bigraphical models of containers

DBCChecker is based on the abstract bigraph based model of container based systems presented in (Burco, Miculan, Peressotti, ACM SAC 2020), and the JLibBig implementation of bigraphs.



JSON Bigraph Format (JBF)

Based upon the standard JSON *Graph* Format.

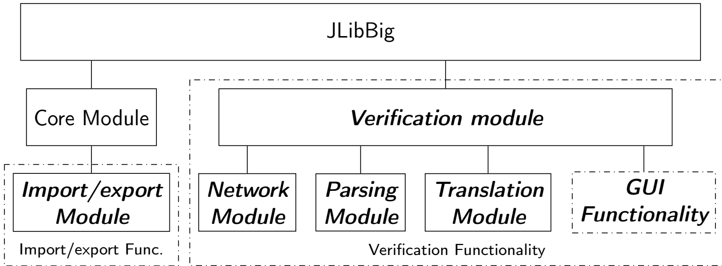
Uses **metadata** objects to describe the signature and other specific informations of directed bigraphs.

An extension to the JBF specification is needed to describe the properties that do not fit in JGF.

```
1  {
2    "graph": {
3      "nodes": {
4        "NodeName": {
5          "metadata": {
6            "type": "type"
7          },
8          "label": "label"
9        }
10     },
11     "edges": [
12     {
13       "source": "sourceNode",
14       "relation": "relation",
15       "target": "targetNode",
16       "metadata": {
17         "portFrom": "portFrom",
18         "portTo": "portTo"
19       }
20     },
21     {
22       "source": "sourceNode",
23       "relation": "relation",
24       "target": "targetNode",
25       "metadata": {
26         "portFrom": "portFrom",
27         "portTo": "portTo"
28       }
29     }
30     ],
31     "type": "type",
32     "metadata": {
33       "signature": [
34       {
35         "name": "name",
36         "arityOut": 1,
37         "arityIn": 1
38       }
39       ]
40     }
41   }
42 }
```

DBCChecker: tool architecture

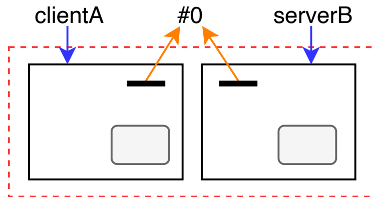
The Verification module is the main actor and controller of our system, responsible for the entire verification process.

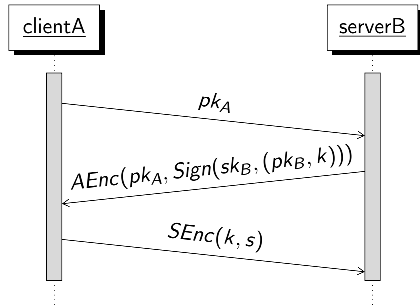


A basic example

Let us consider a very simple handshake protocol between two containers, a client A and a server B, over a shared channel.

Global property to check: confidentiality of s.



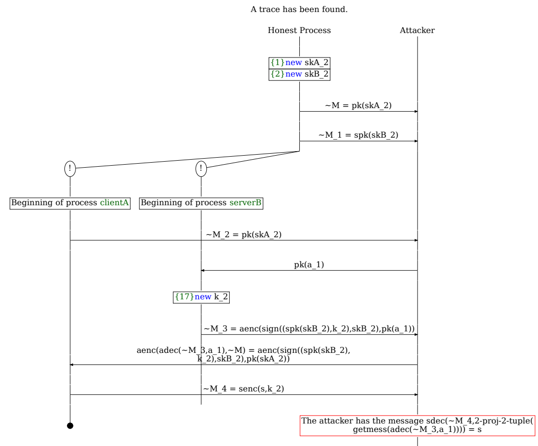


A basic example: contracts

```
1     "clientA": {
2       "metadata": {
3         "type": "node",
4         "control": "1on0",
5         "params": ["pkA:pkey", "skA:skey", "pkB:spkey"],
6         "behaviour": "!(out (#0+, pkA);
                 in (#0+, x : bitstring);
                 let y = adec(x, skA) in
                 let (=pkB, k : key) = checksign(y, pkB) in
                 out (#0+, senc(s, k)))." ,
7         "attribute": ""
8       },
9       "label": "clientA"
10    }
```

```
1     "serverB": {
2       "metadata": {
3         "type": "node",
4         "control": "1on0",
5         "params": ["pkB:spkey", "skB:sskey"],
6         "behaviour": "!(in(#0+, pkX : pkey);
                 new k : key;
                 out(#0+, aenc(sign((pkB, k), skB), pkX));
                 in(#0+, x : bitstring);
                 let z = sdec(x, k) in 0 ).",
7         "attribute": ""
8       },
9       "label": "serverB"
10    }
```

A basic example: results

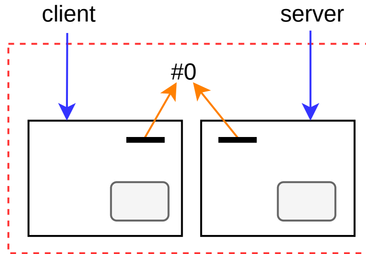


A slightly more advanced example

Two containers are communicating over a *private* channel.

Global property to check: confidentiality of *data*.

The system is secure: the network is internal.

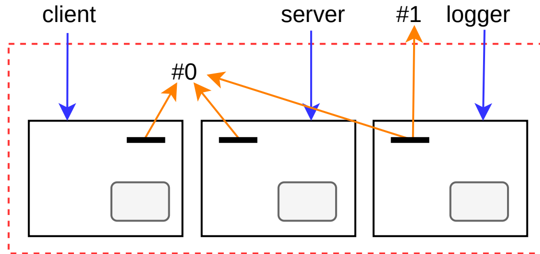


A slightly more advanced example

Two containers are communicating over a *private* channel.

Global property to check: confidentiality of *data*.

The system is secure: the network is internal. However, if we add e.g. a logger the property may not be preserved.



A slightly more advanced example: contracts

```
1  "client": {
2    "metadata": {
3      "type": "node",
4      "control": "1on0",
5      "properties": {
6        "params": [],
7        "behaviour": "new
               data:bitstring; out(#0-,
               data).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "client"
13 },
```

```
1  "server": {
2    "metadata": {
3      "type": "node",
4      "control": "1on0",
5      "properties": {
6        "params": [],
7        "behaviour": "in(#0-,
                data_received:bitstring).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "server"
13 },
```

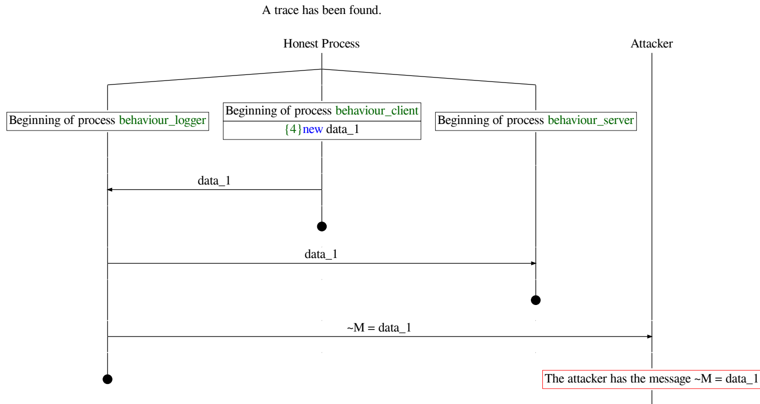
A slightly more advanced example: contracts

```
1  "client": {
2    "metadata": {
3      "type": "node",
4      "control": "1on0",
5      "properties": {
6        "params": [],
7        "behaviour": "new
               data:bitstring; out(#0-,
               data).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "client"
13 },
```

```
1  "server": {
2    "metadata": {
3      "type": "node",
4      "control": "1on0",
5      "properties": {
6        "params": [],
7        "behaviour": "in(#0-,
                data_received:bitstring).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "server"
13 },
```

```
1  "logger": {
2    "metadata": {
3      "type": "node",
4      "control": "2on0",
5      "properties": {
6        "params": [],
7        "behaviour": "in(#0-,
                data_toLog:bitstring);
                out(#0-, data_toLog);
                out(#1+, data_toLog).",
8        "events": [],
9        "attribute": ""
10     }
11   },
12   "label": "logger"
13 },
```

A slightly more advanced example: results



What we have done

Introduced JBF, a specification language for containerized architectures based on bigraphs.

Introduced DBCChecker, a tool to verify security properties of systems obtained by composition of containers.

DBCChecker builds a model of the overall system, which can be verified in ProVerif, to check the satisfaction of the required properties.

Future work

Extending the specification language to support other properties: e.g. temporal logics, other backends...

Simplifying the user interaction with the system: the implementation of a GUI could be a major step towards a complete and user-friendly toolkit.

Integrate the system with a network discovery tool, in order to simplify the process of modelling and verifying large and dynamic containers based systems.

Integrate within the SECCO pipeline.

Thanks for your attention

Try DBCChecker on GitHub

Marino Miculan
University of Udine & Ca' Foscari University of Venezia
marino.miculan@uniud.it