Supporting Secure Coding with RefactorErl[1]

Brigitta Baranyai, István Bozó$^{[0000-0001-5145-9688]}$**,**
Melinda Tóth$^{[0000-0001-6300-7945]}$

ELTE, Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers
`ljcy93@inf.elte.hu, toth_m@inf.elte.hu, bozo_i@inf.elte.hu`

Abstract

In the era of the Internet there is a continuous demand for distributed systems which should serve thousands of requests on a daily basis. But with this growing demand companies have to face a growing number of cyber threats as well which can not only harm their customers in the form of data theft or data loss but their own reputation as well. In order to improve the security of the systems, there are several standards (CERT, OWASP's Application Security Verification Standard) and static analyser tools (CodeChecker, SpotBugs, SonarQube, Fortify) to achieve this goal. Unfortunately, these tools only cover popular programming languages like C++, Java, Python or TypeScript. In this paper, we identify secure coding issues for the programming language Erlang, and group them into five categories. We introduce static analysis algorithms to identify the presented issues. We also present a prototype for supporting secure coding with RefactorErl which helps identifying vulnerabilities like OS injection, cryptography or atom exhaustion related attacks in Erlang systems.

1 Introduction

As the programming industry matured, quality assurance became more and more important. Certainly a lot has changed since Erlang was initially introduced in the 80s. For any language to remain relevant it is a necessity to adjust to these new requirements imposed by the industry. It is important to try to close this gap that currently exists in the domain of static security analysers in order to keep Erlang as relevant as it is today and to possibly increase its popularity in the programming industry.

The change of the original concept of running Erlang on protected hosts, and the human factor (e.g. inexperienced programmers, the lack of knowledge about a specific application) of the development process makes it necessary to consider secure coding during the development of Erlang applications as well.

The main contributions of this paper is providing a tool for the Erlang community to help identifying security vulnerabilities in an early phase of the development process, therefore improving the security level of their Erlang applications. We defined checkers to identify the source code fragments that violates the security rules. The checkers were defined using a set of static source code analyses (e.g. data-flow, call-graph information) provided by RefactorErl.

The rest of the paper is structured as follows. In Section 2 we introduce the problem of secure coding. In Section 3 we present Erlang security vulnerabilities we identified based on the language's idiosyncrasies. Section 4 describes RefactorErl, the static analysis tool developed by a research

[1]The research has been supported by the project "Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein (Integrated program for training new generation of researchers in the disciplinary fields of computer science) ", No. EFOP-3.6.3-VEKOP-16-2017-00002. The project has been supported by the European Union and co-funded by the European Social Fund. The research is part of the "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme no. 2020-4.1.1.-TKP2020 (National Challenges Subprogramme) funding scheme.

group at Eötvös Loránd University and its semantic query language which provides an ideal base for implementing security checkers and make it available to developers. In this section we also outline the algorithm for identifying the potential threats of an Erlang system. Section 5 demonstrates the framework on some open-source Erlang projects. Finally, Section 6 discusses related work and Section 7 concludes the paper and present future ideas for improving the tool.

2 Secure Coding

In software development one of the most important concepts is code quality. Besides the maintenance cost of a not well written code base, there is also a higher risk of introducing bugs during the development phase. In order to maintain code quality, developers should consider the following aspects: availability, fault tolerance, reliability and early identification of possible vulnerabilities. By using static analyser tools and continuous code reviews one can reduce the chance of introducing bugs but in some cases (permissive language syntax, toolset) these methodologies are simply not good enough to keep the software completely bug-free.

Many distributed systems written in the popular programming languages like C, C++, Java are using the concept of Shared Everything [1] which heavily relies on the idea of resource sharing (memory, processing unit or data source). While it can have a positive impact on performance, there are downsides as well, such as the higher risk of introducing vulnerabilities coming from the inappropriate data encryption, race conditions or an error in a subcomponent leading to complete system failures. These languages also often encourage mutable data types which often leads to inconsistent state. In addition to that, various language specific idiosyncrasies [2] such as evaluation related side effects, casting, overloading or the permissive language syntax like optional braces for control statements can cause further issues.

Based on the Application Security Verification Standard [3] the most well-known vulnerabilities are coming from inappropriate input validation, output encoding, authentication and password management, session management and access control, data protection, system configuration, database security, file or memory management.

Erlang embraces the concept of Shared Nothing which means that the processes of the distributed network run isolated with their own resources. In addition to this, immutable data structures, pure functions, modularity and having fault tolerance as a core language concept also greatly contribute to the hardening of the system against security attacks.

Despite the mentioned language concepts and features, many of the above vulnerabilities (file management, input validation, authentication management, etc.) can still affect Erlang due to its original design.

3 Secure Coding in Erlang

Erlang [7] was originally designed for implementing telecommunication systems. Back then these systems demanded requirements [4] like reliability, availability, supporting large numbers of concurrent activities, distributed operations and having fault tolerance against software or hardware errors. Nowadays these requirements are also applicable in other areas [5] like the social media, the financial industry or the Internet of Things. For example, in the financial industry, the considerations highlighted above are even more important. While the loss of a message or the inaccessibility of a mobile device for a short time is not a critical issue, a failed financial transaction or downtime of a trading system can have serious consequences.

As web applications came more and more popular, the distributed systems [6] were redesigned from running on multiple core computers to the geographical distributed, web based grids. Due to the fact that Erlang was mainly designed for private and secured networks, it does not provide a safe communication layer which would be necessity against security attacks.

In the following subsections we are describing some of the well-known issues [8, 9] to showcase the vulnerability types that can be identified with RefactorErl [25, 26] as of now. We grouped the identified issues into five categories:

- Interoperability mechanism related vulnerabilities
- Concurrent programming related issues
- Distributed programming related issues
- Injections
- Memory overload related attacks

3.1 Interoperability Mechanism Related Vulnerabilities

Erlang supports interoperability with modules written in other languages, such as C or Java. There are three ways to do this: using Erlang ports where the external program runs outside of the Erlang VM, running as dynamically loaded libraries (`erl_ddll`) within the Erlang's runtime environment, or through Native Implemented Functions (NIF). These features should be used cautiously. Without proper input validation the loaded files can cause undefined behaviour. Errors coming from the port drivers or NIF functions can cause further instability or even the termination of the Erlang runtime system.

3.2 Concurrent Programming Related Issues

First class support for concurrent programming is part of the language design since the beginning, and it is one of the reasons Erlang is popular in this domain. Besides its advantages, there are some concepts which can cause undefined behaviour in the system. These can be confusing to developers just getting to know the language, so it is beneficial to show them. For example, it can lead to instability if the target process terminates sooner than the `link/1` function is called to begin a connection attempt. A much safer option is to use the `spawn_link/1-4` function for creating and connecting the processes with each other due to its atomic operations.

Although in most cases the isolated manner of the processes save applications from race conditions, process starvation or deadlocks, changing the priority of processes can bring the process scheduler into an undeterministic state.

In addition to the above issues, the usage of ETS tables can cause further unsecure behaviour. Although data modification happens in an atomic and isolated way, the same guarantee does not apply to table traversal when objects are changed, inserted or removed in parallel to the traversal. These operations can result in inconsistency of the table. In order to avoid this and guarantee consistency, the table should be fixed by calling the `safe_fixtable` function before using traversal operations like `first/1, next/1, match/1-3, select/1-3`.

3.3 Distributed Programming Related Issues

As more and more people use the Internet for their daily jobs and personal needs, there is an increasing demand for distributed systems in many industries like telecommunication, finance, social media and e-commerce. To ensure that these interactions are secure, encryption has to be enabled between the nodes of the distributed network. Since at the beginning Erlang was mainly used on private and secured networks, security considerations such as hardening the standard library did not get as much emphasis as is customary for languages actively used in less secure environments.

Due to this fact there is a high chance of potential security issues related to distributed programming in Erlang. Since the Erlang VM only supports primitive access control, the connected remote nodes gain access not only to the distributed node but also to the machine itself on which the node is running. In order to reduce the chance of giving unintended access to an unsafe node, developers should be cautious of using the network kernel related functions. This includes `allow/1` or `connect_node/1` from the `net_kernel` module which allow connection of nodes into the network.

Some of the available options to support backward compatibility with legacy systems are considered unsafe. To prevent Man-in-the-middle attacks [12], it is recommended to avoid using the following SSL-3.0 and TLS-1.0 protocol configuration options for the communication over sockets via the `ssl` module:

- {`padding_check, false`}: turns off padding check which makes Erlang programs unsecure against POODLE attacks [10].
- {`beast_mitigation, disabled`}: turns off BEAST mitigation which enables the chance of the BEAST attack where the attacker wants to acquire a cookie of a user session.
- {`fallback, true`}: enables downgrade of TLS protocols to a less secure protocol.
- {`dh, given with any binaries`}: necessary for Diffie-Hellman key exchange which is not supported anymore above TLS-1.3.
- {`dhfile, given with any file names`}: similarly to the previous settings, this is not supported anymore above TLS-1.3.

The `crypto` module gives access to low level cryptography primitives presented in the OpenSSL libraries. Using these functions is discouraged in favor of higher level APIs to prevent programming errors. With the evolution of the OpenSSL package some of the functions became obsolete, for example the `block_encrypt/3/4`, `cmac/3/4`, etc. and therefore not recommended to be used anymore.

3.4 Injection

Based on the OWASP Top 10 Application Security Risks list, injection is one of the most common attack types which can occur in Erlang via OS commands (`os:cmd/1/2, os:putenv/2`), file related operations (`file:eval/1/2, file:script/1/2`) or dynamically compiled and loaded program code (e.g.: `compile:file/1/2, code:atomic_load/1, code:load_binary/3`). Without proper input validation attackers can run malicious code or launch DOS attacks [11].

3.5 Memory Overload Related Attacks

In Erlang atom values are stored in a global atom table. It has a fix size which can be set by developers by passing the size at startup of the VM. Atoms created during runtime are appended to this table but unused values are never removed. Reaching the capacity of the table while adding a new value causes the VM to crash. In order to prevent atom exhaustion, developers should use dynamic atom creation related functions such as (`erlang:list_to_atom/1`, `erlang:binary_to_atom/1/2`) and xml parsing related functions (`xmerl_scan:file/1/2`, `xmerl_sax_parser:file/2`) carefully and with proper input validation. In case of xml parsing, Erlang provides event handlers for preventing the internal or external entity expansion (`internalEntityDecl` or `externalEntityDecl`).

4 The Security Checker of RefactorErl

RefactorErl provides thorough syntactic and semantic analysis capabilities making it ideal for implementing security checkers for the vulnerabilities mentioned so far. This section describes the static analysis algorithms of the checkers to identify the related vulnerable code fragments and a working prototype for implementing such security checkers.

4.1 RefactorErl

RefactorlErl [16, 15, 13] is a static source code analyser and transformation tool developed by a research group at Eötvös Loránd University, Budapest, Hungary. Its main purpose is to support source code transformations without changing behaviour. But with its continuous development now it also helps in understanding huge code bases, their maintenance or even investigating bugs by tracing back their original values of the expressions. It runs on Linux, OS X and Windows. Developers have the opportunity to use it in a way that is most convenient for them: it integrates well with editors like Emacs, Vim, Visual Studio Code and Eclipse. One can use it through a web interface or from the command line through interactive shell. The analyses of the tool can be run on the entire project or specific files, on the source code or on the binary code.

RefactorErl represents the Erlang source code in a Semantic Program Graph (SPG) [17]. The base of the SPG is an Abstract Syntax Tree (AST) containing the lexical and syntactic elements calculated during the initial analysis. Then various semantic analysers (function analyser, data-flow analyser, record analyser, variable analyzer, etc.) incrementally add static semantic information to the AST and that results our SPG. We are querying information from the SPG during the vulnerability checking analysis.

4.2 Detecting vulnerabilities with RefactorErl

The algorithm for identifying the potential threats of an Erlang application is similar for all the attack types. First, we determine the function call locations which are associated with unsecure operations like OS calls. Then we select the function parameters that can be associated with potential vulnerabilities and run dataflow analysis on them. Developers can provide a whitelist of functions for the input validator. Any function encountered which is on the whitelist is considered reliable, and therefore passing the results of these functions as parameters is assumed safe. Finally, we inspect the sensitive parameters of the analysed attack types by tracing back their origin. In most of the cases a parameter with an unknown source could mean a potential risk for the application, therefore such variables are automatically flagged as unsecure. We will use the Algorithm 1 to illustrate how we can detect the OS injection with RefactorErl.

Algorithm 1: The algorithm for detecting OS injection

```
Function get_calls_for_os($Fun$)
Matchers $\leftarrow [\{os, [cmd, putenv]\}]$
OsFuns $\leftarrow get\_calls\_for(Fun, Matchers)$

FunParamTuples $\leftarrow new\_list$
for $O \in OsFuns$ do
    Param $\leftarrow get\_expr\_params\_for(O, Matchers, get\_all\_children(O))$
    $add\_to\_list(\{O, Param\}, FunParamTuples)$
end

DataFlowTuples $\leftarrow get\_origins(FunParamTuples)$
NonSafeDataFlowTuples $\leftarrow get\_unsafe\_funs(DataFlowTuples)$

FunsWithNoBody $\leftarrow get\_funs\_no\_body(NonSafeDataFlowTuples)$
ExportedFuns $\leftarrow get\_exported\_funs(NonSafeDataFlowTuples)$

return $merge(FunsWithNoBody, ExportedFuns)$
```

To store the unsecure operations essential for the analysis we are using macros in the format illustrated below:

```
-define(xml_usage, [{xmerl_scan, [file, string]},
                    {xmerl_sax_parser, [file, stream]}]).
```

The analysis starts with the function passed in as a parameter. Based on its Semantic Program Graph we are collecting all the expressions which refer to the examined attack type, in our case the `cmd` and `putenv` functions from the `os` module.

In order to find the sensitive parameters of the functions collected in the previous phase, we are calling the `get_expr_params_for/3` function from the Algorithm 2 recursively. To identify the function calls we are using a switch statement. If the input of the recursive call is an expression with result, we identify it as a type of `match_expr`. In this case we are picking the right side of the equation which we can further analyse with the help of the Semantic Program Graph. The subgraph we get as a result of the analysis will be the input of our next recursive call. We continue this process until we get an expression with the type of `application` which means a function call in the SPG. We provide the path to the unsecure parameters with higher order functions which we call when we execute the block of the `application` case. In other cases we simply work with both the left and the right subgraphs of the graph.

Algorithm 2: Querying the sensitive parameters

```
Function get_expr_params_for($Expr, Matchers, FunDef$)
ExprType ← $get\_expr\_type(Expr)$

switch $ExprType$ do
    case $application$ do
        SubExpressions ← $get\_sub\_expressions(Expr)$
        MatchingExpressions ← $new\_list$
        for $SubExpr \in SubExpressions$ do
            if $member(SubExpr, Matchers)$ then
                $add\_to\_list(SubExpr, MatchingExpressions)$
            end
        end
        Params ← $FunDef(MatchingExpressions)$
    end
    case $match\_expr$ do
        Params ← $get\_expr\_params\_for(get\_2nd\_child(Expr), FunDef)$
    end
    otherwise do
        Params ← $get\_all\_children(Expr)$
    end
end

return Params
```

With the help of the dataflow analysis [14, 16] we are tracing back the origin of the data appearing in expressions and variables in order to collect information about its potential value at a given point of the program execution or about the dependencies between the expressions. With this process we can track the occurrences of data and in our case the fact that what is the source of the examined parameter. To do so we run a ”reach” type dataflow analysis on the sensitive parameters collected in the previous phase. This gives us all the expressions with their results containing the examined parameter which is accessible from the analysed expressions.

In special cases the process can differ from the previously described, for example, if the result of a function call is stored in a `tuple` data type like illustrated below:

```
{ok, X} = io:read("Read: "),
os:cmd(X).
```

Following the previously described analysis, we would miss the `io:read/1` function call, therefore the dataflow analysis should be run on one level higher, namely on the `{ok, X}` `tuple` expression. Therefore we have added an extra algorithm for embedded argument handling (Algorithm 3).

Algorithm 3: Dataflow analysis ran for sensitive parameters

```
Function get_origins($FunParamTuples$)
DataFlowTuples ← $new\_list$
for $\{Expr, Param\} \in FunParamTuples$ do
    Df ← $data\_flow\_reach(Param)$
    MergedDf ← $merge(Df, data\_flow\_reach(sel\_back(Df)))$
    OuterDf ← $merge(MergedDf, data\_flow\_reach(sel\_back(MergedDf)))$
    $add\_to\_list(\{Expr, \{Param, OuterDf\}\}, DataFlowTuples)$
end
return DataFlowTuples
```

By setting the `safe_funs` environment variable in the interactive shell of RefactorErl, developers can specify a list of functions they want for the checker to assume to be safe. This feature can be used to address certain false positives and to specify functions that take care of input validation.

```
ri:addenv(safe_funs, [{test_mod, [safe_fun]}]).
```

Using the dataflow analysis of the previous phase, we iterate through all the results and mark those safe where any of the elements of the data flow references at least one of the functions listed in `safe_funs`. All the parameters marked safe can be ignored for the rest of the analysis.

Algorithm 4: Identification of the usage of the return values of "bodyless" functions

```
Function get_funs_without_body($Tuples$)
ExprWithUnknownParams ← $new\_list$

for $\{Expr, \{Param, Df\}\} \in Tuples$ do
    FunBodyList ← $get\_function\_body(get\_funs(Df))$

    for $Body \in FunBodyList$ do
        if $is\_empty(Body)$ then
            $add\_to\_list(Expr, ExprWithUnknownParams)$
        end

        else
            Df ← $data\_flow\_reach(Param)$
            if $member(Body, Df)$ then
                $get\_funs\_without\_body(\{Expr, \{Param, Body\}\}, ExprWithUnknownParams)$
            end
        end
    end
end

return ExprWithUnknownParams
```

Finally, in the last phase we collect all the expressions which contain a parameter with an unknown origin. In our algorithm we differentiate two sources of the parameters resulted in vulnerability: the ones of an exported function or the ones which are the return values of a function with an unknown body. In both cases the parameters could mean a potential risk for the application, therefore such expressions are automatically flagged as unsecure.

We consider a function "bodyless" when the function body is unknown for the static analyzer. For example the `io:read/1` function is a "bodyless" function which reads the data from the standard input during the execution time. The behaviour of identifying the expressions with an unknown parameter is demonstrated in Algorithm 4 as well.

First we query the body of the functions coming as a result of the previous phase. If the function is unknown for the analyser, we flag it as a potential risk. Otherwise, we're looking for matches in a recursive manner between the result of the dataflow analysis and examined function body. In case of a match, we consider the parameter as safe.

The Algorithm 5 demonstrates the identification of parameters coming from exported functions. First we lookup the function which contains the examined expression and if that function is a member of the list of the exported functions. If it is part of the list, we match the sensitive parameter against the parameter of the exported function. Matching parameters could potentially indicate fragile code.

Algorithm 5: Identifying parameters originated from exported functions

```
Function get_exported_funs($Tuples$)
ExprWithUnknownParams ← $new\_list$
for $\{Expr, \{Param, Df\}\} \in Tuples$ do
    Functions ← $get\_funs(Df)$

    for $Fun \in Functions$ do
        if $is\_exported(Fun)$ then
            ExportedFunParams ← $get\_fun\_params(Fun)$
            if $member(Param, ExportedFunParams)$ then
                $add\_to\_list(Expr, ExprWithUnknownParams)$
            end
        end
    end
end

return ExprWithUnknownParams
```

For some of the identifiable attack types the methodology might diverge. For example in order to find the obsolete functions from the `crypto` module, it is enough to identify the location of the examined function call, it is not necessary to check the parameters they are invoked with. To prevent Man-in-the-middle attacks, we have to check the configuration settings of the SSL-3.0 and TLS-1.0 communication protocol related functions. For identifying ETS table traversal related functions which are executed not in an atomic and isolated way, analyzing the function parameter will not be enough, we also need to know that the table was fixed before the operation. For that we need to trace back that the `safe_fixtable(TableId, true)` call was applied for the table before the traversal.

4.3 Semantic Query Language of RefactorErl

RefactorErl provides a semantic query language [18] to help developers with code maintenance, exploring the code structure and its dependencies or simply to understand an unfamiliar code base. Among other things developers are able to get syntactic and semantic information about Erlang programs by querying the call chains, function calls appearing in expressions, record or macro references, function definitions and some entity related attributes like function arity or expression types.

The units of the query language correspond to the semantic language elements of Erlang, which include the following: files, functions, function parameters, expressions, variables, etc. Each of these units has a set of selectors and properties. Developers can reveal the relationship between the entities by running queries using these selectors. The result of these queries will be the elements which are conforming to the given requirements. Properties give some information about the characteristics of the given entity, like the name or arity of the function. The result of the selectors can be filtered by property related restrictions, logical expressions or comparison.

The following examples demonstrates the query language. Let us say we wanted to lists all functions that are longer than 80 characters, then we can write a query like this:

```
mods.funs[max_length_of_line > 80]
```

More complex queries can be constructed as well. The following one lists all the function calls of the `bar` function from the `foo` module which first argument might has the value `error`:

```
mods[name=foo].funs[name=bar]
   .refs[.param[index=1]
       .origin[type=atom, value=error]]
```

4.4 Currently Available Checks

To cover the vulnerabilities presented in the previous sections we defined the checks in RefactorErl listed in Table 1.

Selectors	Short description
unsecure_calls	Lists all the possible vulnerabilities
unsecure_interoperability	Lists interoperability related weaknesses
unsecure_concurrency	Identifies concurrency related issues
unsecure_os_call	Checks for OS injection
unsecure_port_creation	Identifies port creation related issues
unsecure_file_operation	Lists unsecure file handling
unstable_call	Shows possible atom exhaustion
nif_calls	Identifies unsecure NIF calls
unsecure_port_drivers	Lists the unsecure ddll usage
decommissioned_crypto	Lists the legacy functions from crypto module

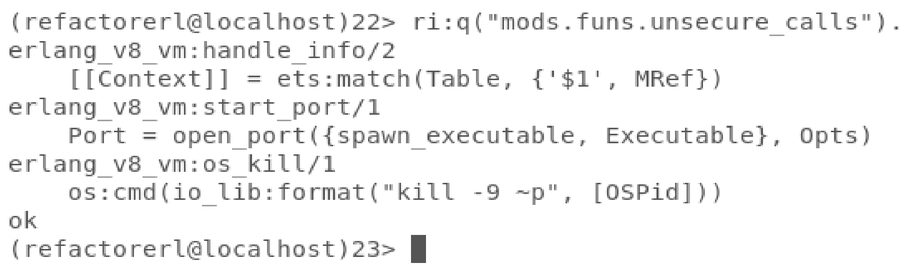
Selectors	Short description
unsecure_compile_operations	Shows unsecure compile/code loading related operations
unsecure_process_linkage	Lists unsecure process linkage
unsecure_prioritization	Identifies unsecure process prioritization
unsecure_ets_traversal	Lists unsecure ETS traversal
unsafe_network	Checks for unsecure kernel related operations
unsecure_xml_usage	Identifies unsecure xml parsing
unsecure_communication	Lists unsecure communication related settings

Table 1: The available security checks in RefactorErl

These checks are available through the selectors that work on functions. The result of these queries is a set of expressions containing the discovered vulnerabilities. Example queries:

```
mods[name=foo].funs[name=bar].unstable_call
mods[name=foo].funs[name=bar].unsafe_network
mods[name=foo].funs[name=bar].unsecure_ets_traversal
```

Developers can use the security checker through the command line tool, like shown in Figure 1.



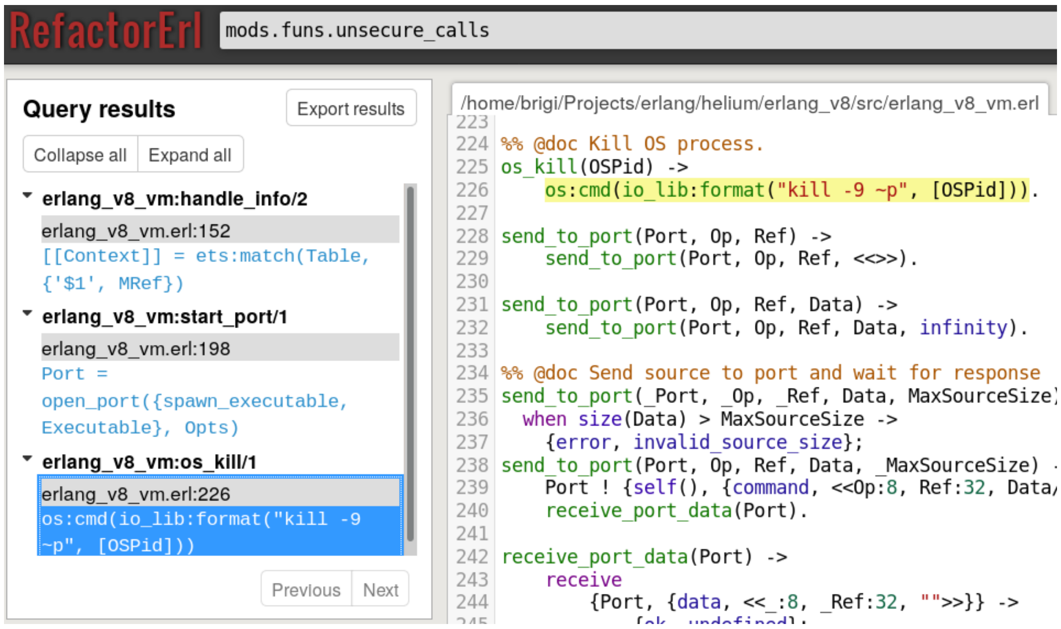
```
(refactorerl@localhost)22> ri:q("mods.funs.unsecure_calls").
erlang_v8_vm:handle_info/2
    [[Context]] = ets:match(Table, {'$1', MRef})
erlang_v8_vm:start_port/1
    Port = open_port({spawn_executable, Executable}, Opts)
erlang_v8_vm:os_kill/1
    os:cmd(io_lib:format("kill -9 ~p", [OSPid]))
ok
(refactorerl@localhost)23>
```

Figure 1: Select unsecure calls through the RefactorErl interactive shell

They also have the opportunity to select the unsecure expressions through RefactorErl's web-based interface where they can not only see the context of the listed vulnerabilities but they can also navigate easily among them (Figure 2).

5 Experimenting with the framework

To demonstrate the effectiveness of our solution, we selected a few open-source projects available on github. Taking into account that we are performing static analysis that may take time on large



mods.funs.unsecure_calls

Query results

```
erlang_v8_vm:handle_info/2
  erlang_v8_vm.erl:152
  [[Context]] = ets:match(Table,
  {'$1', MRef})
erlang_v8_vm:start_port/1
  erlang_v8_vm.erl:198
  Port =
  open_port({spawn_executable,
  Executable}, Opts)
erlang_v8_vm:os_kill/1
  erlang_v8_vm.erl:226
  os:cmd(io_lib:format("kill -9
  ~p", [OSPid]))
```

/home/brigi/Projects/erlang/helium/erlang_v8/src/erlang_v8_vm.erl

```

%% @doc Kill OS process.
os_kill(OSPid) ->
    os:cmd(io_lib:format("kill -9 ~p", [OSPid])).

send_to_port(Port, Op, Ref) ->
    send_to_port(Port, Op, Ref, <<>>).

send_to_port(Port, Op, Ref, Data) ->
    send_to_port(Port, Op, Ref, Data, infinity).

%% @doc Send source to port and wait for response
send_to_port(_Port, _Op, _Ref, Data, MaxSourceSize)
  when size(Data) > MaxSourceSize ->
    {error, invalid_source_size};
send_to_port(Port, Op, Ref, Data, _MaxSourceSize) -
    Port ! {self(), {command, <<Op:8, Ref:32, Data/
    receive_port_data(Port).

receive_port_data(Port) ->
    receive
        {Port, {data, <<_:8, _Ref:32, "">>}} ->
```

Figure 2: Select unsecure calls through RefactorErl's web interface

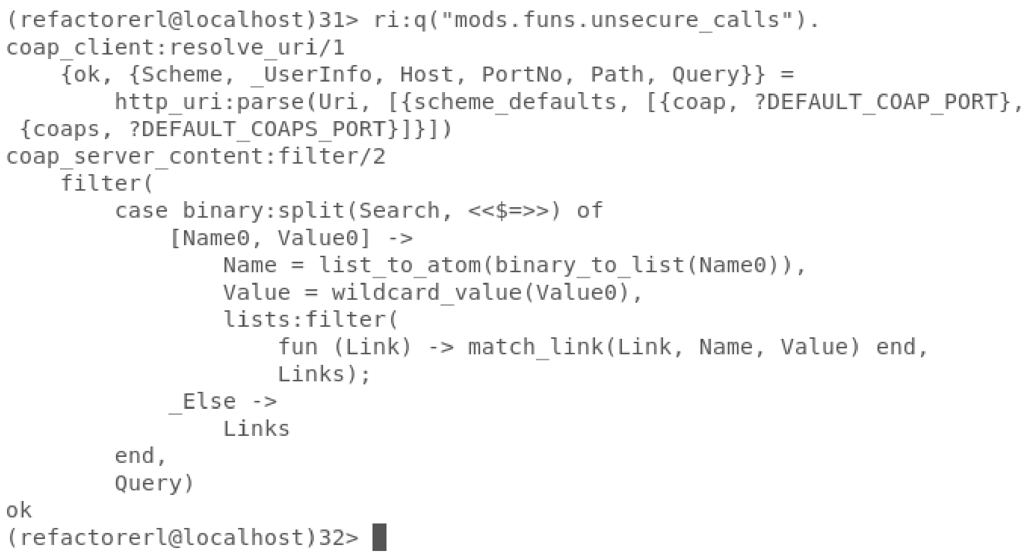
projects, we selected projects with different sizes and also domains. [2] Table 2 shows the number of found vulnerabilities and their types. We found that the majority of the candidates were ets usage and atom creation related [3], but the typical injection-like candidates were found as well. One might realize that the size of the application does not correlate to the number of found vulnerabilities.

Project	Num.	Type of the vulnerabiliy	LOC	Time (sec)
helium/erlang_v8	3	*unsecure_ets_traversal (1), unsecure_port_creation (1), unsecure_os_call (1)*	302	1
helium/miner	7	*unsecure_os_calls (3), unstable_call (4)*	6236	15
heroku/logplex	15	*unsecure_ets_traversal (11), unstable_call (4)*	10634	25
k2informatics/ranch	7	*unsecure_ets_calls (7)*	3052	7
relayr/gen_coap	2	*unstable_call (2)*	2076	6

Table 2: Example results

Let us take a closer look at one of these projects. We identified two atom exhaustion related vulnerabilities in the `relayr/gen_coap` source code, shown in Figure 3.

[2] The extensive evaluation of the results is an ongoing work.
[3] We also examined libraries from the source of the Erlang/OTP and found that the dynamic atom creation is quite common (e.g. the asn library).

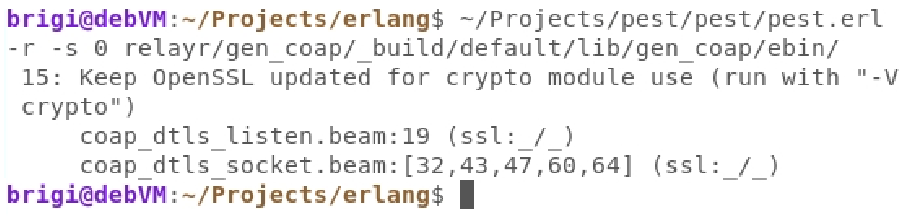


```
(refactorerl@localhost)31> ri:q("mods.funs.unsecure_calls").
coap_client:resolve_uri/1
    {ok, {Scheme, _UserInfo, Host, PortNo, Path, Query}} =
        http_uri:parse(Uri, [{scheme_defaults, [{coap, ?DEFAULT_COAP_PORT},
 {coaps, ?DEFAULT_COAPS_PORT}]}])
coap_server_content:filter/2
    filter(
        case binary:split(Search, <<$=>>) of
            [Name0, Value0] ->
                Name = list_to_atom(binary_to_list(Name0)),
                Value = wildcard_value(Value0),
                lists:filter(
                    fun (Link) -> match_link(Link, Name, Value) end,
                    Links);
            _Else ->
                Links
        end,
        Query)
ok
(refactorerl@localhost)32> 
```

Figure 3: Identified vulnerabilities in the relayr/gen_coap source code

The `resolve_uri` function is an exported function taking a URI's scheme as an input and converting it to an atom by the `http_uri:parse` function. As the URI is taken from an untrusted source, it can lead to atom exhaustion that can cause the Erlang VM to crash. Therefore we list this as a potential risk for the user.

From the available static source code analysis tools only PEST (Primitive Erlang Security Tool) [19] provides similar security inspections. To compare the tools we have run the checks provided by the PEST tool for the `relayr/gen_coap` source code, the result is shown in Figure 4.



```
brigi@debVM:~/Projects/erlang$ ~/Projects/pest/pest/pest.erl
-r -s 0 relayr/gen_coap/_build/default/lib/gen_coap/ebin/
 15: Keep OpenSSL updated for crypto module use (run with "-V
 crypto")
    coap_dtls_listen.beam:19 (ssl:_/_)
    coap_dtls_socket.beam:[32,43,47,60,64] (ssl:_/_)
brigi@debVM:~/Projects/erlang$ 
```

Figure 4: Identified vulnerabilities in the relayr/gen_coap source code with PEST

The identified vulnerabilities are related to the usage of the `ssl` module. The differences lie in what methodologies the tools are using. Since PEST is based on simple text search it will show every occurrence of the function calls of the `ssl` module regardless of the parameters passed to these functions. PEST does not report the `list_to_atom` calls, as in the source code the module name is omitted, therefore the `erlang:list_to_atom` rule does not match.

6 Related work

The early detection of potential vulnerabilities is recognised to be important in the software development industry. There are a number of static analysis tools to help developers indentify such issues like SonarQube [22], SpotBugs [21] and CodeChecker [20]. Most of these tools integrate well into the popular IDEs and editors such as: IntelliJ, Visual Studio Code and Eclipse, and can be run as part of a Continuous Integration (CI) pipeline.

This topic is becoming more and more important in the Erlang community which is shown by the increasing number of papers and discussions that try to raise security awareness [23, 24]. PEST [19] is one of the already available tools to aid developers identifying security issues. It analyzes the source code to detect interoperability related functions like NIF or port driver creation calls, the usage of OS calls or the decommissioned functions from the `crypto` module. The tool can be run as an escript from command line and provides some extra settings like expanding the checks for the dependencies as well or setting the severity of the issues to filter out the less significant vulnerabilities. The main downside of the tool that it is based on simple text search, so there is a high chance for listing false positive results. Based on the thorough static semantic analysis framework of RefactorErl we were able to produce more accurate results.

There is also another prototype [24] based on RefactorErl which identifies similar vulnerability types. This work focuses on providing a systematic way to identify "trust zones" in Erlang applications at the design phase. The potential benefit of this approach is that security verification can be simplified by using these "trust zones" as the basis of the static analysis.

7 Conclusion and future work

Secure coding is a more and more important topic in the era of the Internet. As the number of user interactions and network traffic increases, we see a significant grow in the number of cyber-attacks, which can not only harm customers, but it affects the reputation of companies as well.

In this paper, we presented some of the Erlang security vulnerabilities we identified during our research, which are mainly connected to the language's interoperability mechanisms and the way concurrent and distributed programming was designed. We showcased the work we did in RefactorErl to support secure coding in Erlang.

Although we are using first-order data-flow analysis that takes into account some runtime context information, our results can be improved. We will focus on reducing false-positive cases in the future using heuristics, dynamic information and higher-order data-flow analysis. However, these may increase the runtime of our algorithms.

Our tool can be extended with additional rules to identify race conditions, obsolete cypher algorithms from the `crypto` module, security level related settings, configurable analysis to further enhance the user experience.

It is also possible to add checkers for other programming languages like Elixir which leverage the Erlang VM. Since modules written in Erlang are available to be used in Elixir, there is a potential to reuse the existing module specific checkers with minimal effort.

References

[1] Kenji Rikitake. *Shared Nothing Secure Programming in Erlang/OTP.* In IEICE Tech. Rep., volume 114 of IA2014-11, ICSS2014-11, pages 55–60, Hyogo, June 2014, Takikawa Memorial Hall, Kobe University (ICSS, IA).

[2] E. Jaeger and O. Levillain. *Mind Your Language(s): A Discussion about Languages and Security.* In 2014 IEEE Security and Privacy Workshops, pages 140–151, 2014.

[3] The OWASP Foundation. *Application Security Verification Standard.* `https://owasp.org/www-project-application-security-verification-standard/`, 2020.

[4] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* A Dissertation submitted to the Royal Institute of Technology in partial fulfilment of the requirements

for the degree of Doctor of Technology The Royal Institute of Technology Stockholm, Sweden, 2003.

[5] Francesco Cesarini. *Which companies are using Erlang, and why?*. `https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus.html, 2019.`

[6] Kenji Rikitake and Koji Nakao. *Application security of erlang concurrent system.* 2008.

[7] Francesco Cesarini and Simon Thompson. *ERLANG Programming (1st. ed.).* O'Reilly Media, Inc, 2009. ISBN 0596518188.

[8] Fred Herbert. *Learn You Some Erlang for Great Good!: A Beginner's Guide.* No Starch Press, San Francisco, CA, USA, 2013. ISBN 1593274351, ISBN 9781593274351.

[9] Security Working Group of the Erlang Ecosystem Foundation. *Secure coding and deployment hardening guidelines.* `https://erlef.github.io/security-wg/, 2020.`

[10] Adam Langley. *The POODLE bites again.* `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-15160, 2020.`

[11] Khaled Elleithy, Drazen Blagovic, Wang Cheng and Paul Sideleau. *Denial of Service Attack Techniques: Analysis, Implementation and Comparison.* Journal of Systemics, Cybernetics and Informatics, pages 66-71, 2006.

[12] Rapid7. *MITM Techniques, Detection, and Best Practices for Prevention.* `https://www.rapid7.com/fundamentals/man-in-the-middle-attacks/, 2020.`

[13] RefactorErl. *Static source code analyser and refactoring tool for Erlang.* `https://plc.inf.elte.hu/erlang, 2020.`

[14] Melinda Tóth. *Data flow and dependence analyses for functional languages – Static analysis of Erlang programs.* 2018. `http://hdl.handle.net/10831/40597.`

[15] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. *Refactorerl - Source Code Analysis and Refactoring in Erlang.* In Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2, pages 138–148, Tallin, Estonia, October 2011.

[16] Melinda Tóth and István Bozó. *Static Analysis of Complex Software Systems Implemented in Erlang.* Central European Functional Programming Summer School - Fourth Summer School, CEFP 2011, Revisited Selected Lectures, Lecture Notes in Computer Science (LNCS), Vol. 7241, pp. 451-514, Springer-Verlag, ISSN: 0302-9743.

[17] Zoltán Horváth and László Lövei and Tamás Kozsik and Róbert Kitlei and Anikó Nagyné Víg and Tamás Nagy and Melinda Tóth and Roland Király: Modeling semantic knowledge in Erlang for refactoring, In Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009, pages 7–16, Studia Universitatis Babeş-Bolyai, Series Informatica, Vol. 54(2009) Sp. Issue, 2009

[18] László Lövei, Lilla Hajós and Melinda Tóth. *Erlang Semantic Query Language.* Proceeding of 8th International Conference on Applied Informatics, ICAI 2010, ISBN 978-963-98-94-72-3, pages 165-172.

[19] Michael Truog. *Primitive Erlang Security Tool (PEST).* `https://github.com/okeuday/pest, 2020.`

[20] CodeChecker. *Official documentation of CodeChecker.*
`https://codechecker.readthedocs.io/en/latest/, 2020.`

[21] SpotBugs. *SpotBugs hivatalos dokumentációja.*
`https://spotbugs.readthedocs.io/en/latest/introduction.html, 2020.`

[22] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action.* Manning Publications, 2013, ISBN 978-161-72-90-95-4.

[23] Alexandre Jorge Barbosa Rodrigues and Viktória Fördős. *Towards secure erlang systems.* In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang, Erlang 2018, page 67–70, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358248.

[24] Fördős, Viktória: Secure Design and Verification of Erlang Systems, Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang, pages 31–40, Association for Computing Machinery, New York, NY, USA, https://doi.org/10.1145/3406085.3409011, ISBN 9781450380492, 2020

[25] Brigitta Baranyai: Funkcionális nyelvek és a statikus kódelemzéssel támogatott biztonságos szoftverfejlesztés, Paper at the Student Association Conference, Faculty of Informatics, Eötvös Loránd University, May 2020, Received 2nd prize.

[26] Brigitta Baranyai, Melinda Tóth, István Bozó: Supporting Secure Coding with RefactorErl, Talk at the 19th ACM SIGPLAN International Workshop on Erlang, Virtual Event, 23 August, 2020