Erlang 並行システムでのアプリケーションセキュリティ

力武 健次 [†]　中尾 康二 [†◇]

[†] 独立行政法人 情報通信研究機構 インシデント対策グループ

[◇] KDDI 株式会社 情報セキュリティフェロー

拡張性のある大規模アプリケーションを実現する環境として，並行プログラミング言語 Erlang が普及しつつある．しかし，Erlang を広域インターネット上の攻撃に耐え得るものにするには，より高度なセキュリティ保護機構の追加が必要である．本論文では，Erlang の分散仮想マシン環境上の既存のセキュリティ機構を概観すると共に，並行性とスケーラビリティを維持しつつ広域インターネットでの攻撃に耐え得るにはどのような機能を強化すべきかについての考察と提案を行う．

Application Security of Erlang Concurrent System

Kenji RIKITAKE[†], Koji NAKAO[†◇]

[†] Network Security Incident Response Group (NSIRG), NICT, Japan

[◇] Information Security Fellow, KDDI Corporation

Contact email: `rikitake@nict.go.jp`

The Erlang programming language has been popular for a scalable development of large-scale concurrent application software. The reinforcement of security capability of Erlang is essential to defend the software from wide-area Internet attacks. In this paper, we first survey the current security functionality implemented in the Erlang distributed virtual machine environment. We then evaluate and propose the possible further enhancements to add the protection against wide-area Internet attacks, while retaining the concurrent scalability of the Erlang-based systems themselves.

1 Introduction

Large-scale and complex Web computing has become the social infrastructure of the 21st century. The realization technologies of such large systems have been largely changed to decentralization, as the upper-limit of CPU clocks is to be defined by a thermal limitation of the heat dissipation, and the global concern of efficient energy consumption for the available computational performance. Distributed systems of various scales, from many-core processors on a single computer breadboard to the world-wide computing grid connected by the Internet, have been widely utilized as a part of production systems.

Building a decentralized large-scale system is a difficult task also from the programmer and system architect points of view. More fine-grained granularity of computing processes are demanded to process a large number of simultaneous requests, as the context switching between multiple process entities have become a large overhead. Operating system *processes*, where each of them has their own independent memory spaces, are now being redesigned into *threads*, where each of them now *shares* the execution memory space.

Threads, however, have a security weakness due to sharing the same memory space for different purposes, which often leads to deadlocks and race conditions. Rewriting the iterative code into sharing-conscious thread-based code is also a difficult task. Excessive use of locks will degrade the performance, while critical sections will fail without guaranteeing atomic execution. Stewart [1] describes the profiling mechanism for reducing multi-CPU lock contention for SCTP [2] stack of FreeBSD and MacOS X. While the lock profiling mechanism is already built-in to the kernel, reduction of the locks requires manual data structure redesign by the programmers.

Management, synchronization, and communication of the units of execution (UEs) in parallel and concurrent processing need specific software construction techniques [3, Chapter 6] with specific libraries such as OpenMP [4] and OpenMPI [5]. Those frameworks are for scientific programming based on the iterative programming languages such as C++ and Fortran, and still do not prevent bugs specific to concurrent processing, such as indefinite execution state by shared mutable variables without locks.

To solve the difficulty of concurrent programming, the *Erlang* programming language [6] has been gaining popularity among the programmers and system architects to build a large-scale communication systems. Erlang has the language and system features for concurrent programming among distributed virtual machines (nodes), which can be deployable among multiple CPU cores and a cluster of network-connected hosts. The language design of Erlang does not allow mutable variables, so the errors caused by shared states between the UEs will not happen.

The development of Erlang was started in 1986 for programming a telephone exchange of Ericsson [7], and it has become open-source in 1998 [8]. Erlang are widely used among large-scale Web production systems such as Facebook chat system [9] and Amazon SimpleDB [10].

The communication security of Erlang nodes, however, are basically designed for a firewall-protected internal network. The authentication and encryption of the inter-node communication channels are inadequate for a hostile environment such as those over the wide-area Internet. While the current security functionality of Erlang works well for the nodes as a part of internal service systems, more functionality elements to defend the nodes should be added if the Erlang node networks are formed over the Internet.

The rest of the papers is organized as follows. In Section 2, we explain the features of Erlang to prevent bugs on concurrent programming. In Section 3, we evaluate the known vulnerabilities of Erlang security functionality elements for hardening the distributed processing library based on message passing. And in Section 4, we conclude this paper and propose the future direction of our research.

2 Concurrent programming in Erlang

The programming style of Erlang is functional and declarative, which are vastly different from those of procedural and imperative languages such as C++. Erlang also has the following features specifically designed for concurrent programming as follows [11, 12]:

- Erlang variables are *immutable*; the value can be only *given once*, and no multiple assignment allowed.
- Erlang processes *do not share memory space* with each other; the processes have to interact through explicit message passing.
- Erlang process creation speed is much faster than operating system (OS) processes, and roughly equivalent to that of OS thread creation [*1].
- Erlang language runs on a virtual machine (VM) which takes full control of scheduling and resource management between the processes; the basic function modules are OS-independent.

Single Program Multiple Data (SPMD) is a typical pattern of parallel programming [3, Section 5.4]. In an SPMD code, the same subcode is concurrently executed by multiple UEs with different data.

In Erlang, the SPMD pattern can be implemented in a single function as a *parallel map*. An iterative loop execution pattern in Erlang is described as a list argument and *map* [*2] function, since Erlang does not allow a loop index because it is a mutable variable.

In an SPMD code, the computation order of concurrently-executed subcodes does not affect the result of the code, so long as the each set of given data and results for the subcodes are independent with each other. So one-process-for-one-element parallelization of the map function by Armstrong [11, Section 20.2] is sufficient for executing an SPMD pattern in Erlang, provided the subcode is defined as a function spawned as a process, and that the data for each process is a list.

[*1] Armstrong [11, Section 8.4] shows spawning 20000 processes took an average of 3.5μs/process of CPU time on a 2.4GHz Intel Celeron with 512Mbytes of memory running Ubuntu Linux. This result was 20~30 times faster than the OS process creation speed of FreeBSD 5.3 measured by McGarry [13].

[*2] A *map* function performs the same operation to each member of a list, and returns the list of results whose members are in the same sequence as the given list [14].

```
-module(pispmd).
-import(lists,
        [sum/1, map/2, seq/2]).
-import(nsplit, [npmap/3]).
-export([pitest/2]).

sqinv(X, N) ->
      Y = (X + 0.5) / float(N),
      4.0 / (1.0 + (Y * Y)).

%% P: number of processes
pitest(P, N) ->
 sum(npmap(P,
     fun(X) -> sqinv(X, N) end,
     seq(0, (N-1)))) / float(N).
```

Fig. 1 An example of Erlang SPMD code.

(unit: [s])	P: number of processes		
	1000	10000	100000
single node (Host A)	19.82	7.44	9.92
dual node (Host A+B)	14.83	7.00	21.75

(mean values of 10 trials)

Node specification and test conditions:

- CPU: Intel Core2Duo L2400 (max 1.6GHz)
- clock: 875MHz, frequency lowered by FreeBSD `powerd` to prevent performance degradation by the CPU heat dissipation
- memory: 1.5Gbytes
- Erlang VM is SMP-enabled (2 schedulers/node)
- Erlang native code support (HiPE) disabled
- Host A OS: FreeBSD 6.3-RELEASE
- Host B OS: FreeBSD 7.0-RELEASE
- Hosts connected via 100BASE-TX LAN
- execution time measured by `timer:tc` in μs
- N of `pispmd:pitest` = 1000000

Table 1 Execution time of Fig. 1.

For running an SPMD code in an Erlang node, however, one-by-one parallel mapping often causes too much overhead of process creation, especially the computational load of the subcode is small. To solve this problem, Rikitake [15] writes an Erlang module *nsplit*, which splits an argument list into n sublists and spawns a map function for each sublist so that the number of spawned process is limited to n.

Figure 1 is an example of Erlang SPMD code, for estimating the value of π from trapezoidal numerical integration [*3]. In this example, the original iterative code was written with `map` function. Using an n-process parallel map function `npmap` from the nsplit module, instead of the map function, is *the only required modification* to make the entire processing concurrent from a piece of iterative code.

The nsplit module is based on the function named `rpc:parallel_eval` [*4] to implement concurrent execution across the nodes. This function is a part of *rpc* module [16] of the Erlang Open Telecom Platform (OTP), which provides the framework modules for concurrent processing.

The performance gain of running concurrent Erlang processes on multiple nodes differs as the number of concurrent processes P changes, since the overhead of message passing increases as the number of processes increases.

Table 1 is a test result of executing the code of Fig. 1. The overhead of message passing increases as P increases, while the amount of computation for each process decreases, which is proportional to N/P. This result suggests that tuning the process-related parameters is required to obtain the maximum performance from Erlang concurrent systems.

3 Hardening the Erlang systems

From security point of view, distributed Erlang systems have fundamental weaknesses, and require hardening the message passing facility against possible attacks. In this section, we explain the technical details of vulnerabilities on Erlang

[*3] The first 17 digits of π is $3.1415926535897\underline{932}$. The estimated value of π in Fig. 1 with the equation $\pi = \int_0^1 \frac{4}{1+x^2}dx$ is $3.1415926535897\underline{643}$ when N = 1000000. This N is chosen to compute the closet value for an Intel Core2Duo, which uses IEEE-754-compatible floating point arithmetic units.

[*4] In Erlang, a function in a module is uniquely identified with the full name of `module:function`. In this sense, the full name of `map` is `lists:map` and `npmap` is `nsplit:npmap`.

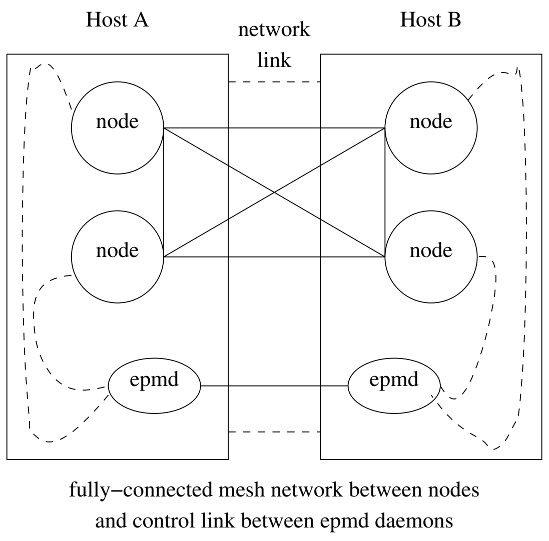
Host A
network
link
Host B
node
node
node
node
epmd
epmd
fully−connected mesh network between nodes
and control link between epmd daemons

Fig. 2 A network of Erlang nodes and hosts.

nodes and epmd processes.

In Erlang/OTP, each node communicates with each other by message passing between the nodes. Figure 2 is an example network between two hosts running four Erlang nodes. Each node uses connection-based links to communicate with each other, forming a fully-connected mesh network, assisted by neighbor-discovery daemons called *epmd* for each host.

On TCP/IP networks, each Erlang node (VM) connects with each other by dedicated peer-to-peer (P2P) TCP connections. The mapping of connection details such as IP addresses and the port numbers, and the symbolic names of nodes in a form `Node@Host`, are maintained by epmd, which also verifies whether a node is alive. The names of hosts are in *short* (without dot) and *long* (with dots such as in domain names) formats. The resolution of IP address from the node names depends on DNS subsystems.

The node authentication to decide whether two nodes are allowed to connect to each other is performed by sharing the same passphrase string called *cookie*. Each Erlang node has a set of the cookie strings for each connecting neighbor node, and uses the strings to perform a challenge-response authentication when connected two nodes start to establish a link as a pair of distributed nodes [*5]. The contents of the data links between the Erlang nodes, however, are not encrypted at all using the default *inet_tcp_dist* driver module, while Erlang allows to choose an alternative driver [17].

Erlang/OTP provides *slave* module as a part of the standard libraries. One of the functions `slave:start` can use the Secure Shell (SSH) as a method of authentication, by invoking a remote Erlang shell through SSH. This function, however, does not encrypt the data links at all. The node mapping is still through epmd and the Erlang nodes still directly connects each other over TCP without encryption on the transport layer.

Links between epmd processes are vulnerable to well-known attacks. The initial port [*6] is well-known, so denial-of-service (DoS) attacks are possible. The TCP/IP sockets are bound to `INADDR_ANY` [*7] and cannot be configured to be bound to a specific interface/address. The epmd links are not encrypted at all, so the port should be isolated by a firewall, or be enclosed in a virtual private network (VPN), to prevent attacks.

Links between Erlang nodes are vulnerable as well as epmd to external attacks, provided the attacker has obtained the information by tapping into the epmd communication channels. While the IP source address and TCP port number range can be configured for each Erlang VM, the choice of port numbers is arbitrary, so controlling access via a firewall is difficult [*8]. Links between Erlang nodes are not encrypted at all as default, unless using an alternative SSL/TLS driver *inet_ssl_dist*, provided as a part of Erlang/OTP code [*9].

[*5] The technical details of data exchange and authentication between distributed Erlang nodes are documented in the following two files in the Erlang source-code tree: `lib/kernel/internal_doc/distribution_handshake.txt` and `erts/emulator/internal_doc/erl_ext_dist.txt`.

[*6] The default value of Erlang epmd TCP port number is 4369, for Erlang Release R12B-3.

[*7] For IP version 4, the value of `INADDR_ANY` is `0.0.0.0`.

[*8] Similar problems occur when handling popular teleconferencing protocols and applications (H.264, Skype, Polycomm, etc.), which allow arbitrary P2P connections with each other.

[*9] As of August 2008 during the preparation of this report, one of the authors Rikitake has not yet obtained successful results of using or even initializing the *inet_ssl_dist* driver. He suspects this is due to the substantial design

Erlang is a user process for the host OS, so the access control must be performed. OS-level virtualization such as FreeBSD *jail* [18] and User-mode Linux [19], or even detailed virtualization software such as VMware ESXi hypervisor [20].

Erlang nodes and epmd have two different set of access control methods, and difficult to configure. The parameters disperse in many configuration options and cannot be centrally managed without an external program.

4 Conclusion and future works

In this paper, we have described the Erlang concurrency and security functions. We conclude that the current Erlang system needs a fundamental redesign for the security functionality, while integration with other security infrastructures such as VPNs and virtual machines may alleviate the vulnerabilities. We explain our future work ideas in the rest of this section.

Erlang provides a practical solution to build a reliable concurrent system, where external security enhancements are adequately provided. The good examples are multi-user communication systems such as ejabberd [21] Jabber/XMPP server and yaws [22] HTTP server. Erlang owes the efficiency to the language features supporting safe and concurrent programming, as shown in Section 2.

The current status of Erlang systems are similar to the situation of the Internet before SSH was invented in 1995; nodes are unconditionally trusting each other, at least for the data channels. The major problem is how to establish a secure P2P link and a set of trusted nodes over an untrusted network such as the Internet. The Erlang distributed node supporting structure is also similar to generic remote procedure call (RPC) subsystems. Development of secure remote configuration mechanism over distributed Erlang node networks such as NETCONF [23] and the SSH implementation [24] is one of the future projects of our interest.

Existing *inet_ssl_dist* module has been a workaround for this problem, but the configuration is complex, and does not work [*10]. Exchanging SSL/TLS certificates is not the only solution; establishment of a pre-shared-key network will be a practical solution to connect large number of hosts over the Internet, and is applicable to Erlang multinode networks.

On a concurrent system, key components of the security algorithms do not scale well. Generating cryptographic random numbers and encrypting/decrypting a cryptographic data stream require keeping the internal states for each stream handled. While current Erlang systems separate the cryptographic functions to another subsystem based on OpenSSL [25], adding a robust support structure in Erlang will soon be necessary. Efficient cryptographic algorithms for concurrent execution is an essential research for making Erlang secure.

We believe that programming languages such as Erlang which help concurrent programming will make the computer communication systems robust and secure. Our future works will include:

- performance measurement of *working inet_ssl_dist* implementation, comparing to the non-cryptographic transport;
- establishing a secure transport mechanism on multiple Erlang nodes over a popular infrastructure of cryptographic communications, such as SSH and Secure HTTP;
- experimenting Erlang cryptographic subsystems on a mixed-OS environment of FreeBSD, Windows, and Linux; and
- IPv6 capability evaluation of Erlang on the security and performance.

Acknowledgements

The first author Rikitake is grateful for discussions on Erlang and other functional programming languages with Hitoshi "amachang" Amano, Ryosuke "voluntas" Nakai, Hirokazu Nishio, and Hirotaka Yoshioka.

References

[1] Stewart, R.: Reducing Lock Contention in a Multi-Core System, *AsiaBSDcon2008 Proceedings*, pp. 115–126 (2008). Tokyo Univer-

update of Erlang/OTP *ssl* module, between the versions 3.1.1 and 3.9.

[*10] See Rikitake's report in *erlang-bugs* mailing list at `http://www.erlang.org/pipermail/erlang-bugs/2008-August/000953.html` for the further details.

sity of Science, Tokyo, Japan, 27-30 March, 2008.

[2] Stewart, R.: Stream Control Transmission Protocol (2007). RFC4960.

[3] Mattson, T. G., Sanders, B. A. and Massingill, B. L.: *Patterns for Parallel Programming*, Addison-Wesley (2005).

[4] OpenMP ARB: OpenMP: Simple, portable, scalable SMP Programming. `http://www.openmp.org/`.

[5] Open MPI Team: Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`.

[6] Armstrong, J.: *Making reliable distributed systems in the presence of software errors*, PhD Thesis, The Royal Institute of Technology, Stockholm, Sweden (2003). `http://www.erlang.org/download/armstrong_thesis_2003.pdf`.

[7] Armstrong, J.: A history of Erlang, *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, ACM, pp. 6–1–6–26 (2007).

[8] Ericsson AB: Open Source Erlang. `http://www.erlang.org/`.

[9] Hoff, T.: New Facebook Chat Feature Scales to 70 Million Users Using Erlang. `http://highscalability.com/new-facebook-chat-feature-scales-70-million-users-using-erlang`.

[10] Ying, C.: What You Need To Know About Amazon SimpleDB. `http://www.satine.org/archives/2007/12/13/amazon-simpledb/`.

[11] Armstrong, J.: *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf (2007).

[12] Armstrong, J., 榊原一矢 (訳): プログラミング Erlang, オーム社 (2008). (This book is Japanese translation of *Programming Erlang* [11].).

[13] McGarry, G.: Benchmark Comparison of NetBSD 2.0 and FreeBSD 5.3. `http://fbim.fh-regensburg.de/~feyrer/NetBSD/gmcgarry/`.

[14] Ericsson AB: Erlang Manual Page for *lists* module. `http://www.erlang.org/doc/man/lists.html`.

[15] Rikitake, K.: nsplit. `http://code.google.com/p/nsplit/`.

[16] Ericsson AB: Erlang Manual Page for *rpc* module. `http://www.erlang.org/doc/man/rpc.html`.

[17] Ericsson AB: How to implement an alternative carrier for the Erlang distribution. Chapter 3 of ERTS User's Guide, `http://www.erlang.org/doc/apps/erts/alt_dist.html`.

[18] The FreeBSD Project: Jails. Chapter 15 of FreeBSD Handbook, `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html`.

[19] Jeff Dike: User-mode Linux. `http://user-mode-linux.sourceforge.net/`.

[20] VMware, Inc.: VMware Web site. `http://www.vmware.com/`.

[21] : ejabberd community site. `http://www.ejabberd.im/`.

[22] : Yaws Web site. `http://yaws.hyber.org/`.

[23] Enns, R.: NETCONF Configuration Protocol (2006). RFC4741.

[24] Wasserman, M. and Goddard, T.: Using the NETCONF Configuration Protocol over Secure SHell (SSH) (2006). RFC4742.

[25] The OpenSSL Project: OpenSSL. `http://www.openssl.org/`.