Security Analysis on InfiniBand Protocol Implementations

Kul Prasad Subedi, Dipankar Dasgupta, Bo Chen
Department of Computer Science
University of Memphis
Memphis, TN 38152, USA
Email: {kpsubedi,ddasgupt,bchen2}@memphis.edu

***Abstract*—The growing popularity of high performance computing has led to a new focus on bypassing or eliminating traditional I/O operations that are usually the bottlenecks for fast processing of large data volumes. One such solution uses a new network communication protocol called InfiniBand (IB) which supports remote direct memory access without making two copies of data (one in user space and the other in kernel space) and thus provides very low latency and very high throughput. To this end, for many industries, IB has now become a promising inter-connect protocol over Ethernet technologies. Ensuring the security of this new protocol is critical since more and more companies are moving towards it.**

To ensure the security of IB, the first step is to have a thorough understand of the vulnerabilities of its current implementations, which is unfortunately still missing in the literature. In this paper, we aim to fill this gap. In particular, we perform a static code analysis as well as protocol testing in order to examine security features in IB architecture from the implementation perspective. While our extensive penetration testing could not find any significant security loopholes; there are certain aspects in both the design and the implementations that need to be addressed. Our focus is in the implementation perspective. Specifically, we found there is a significant use for a number of vulnerable functions (e.g., *memcpy*, *sprintf*, and *char*) as well as obsolete functions (e.g., *memalign*) that we believe should be replaced with alternative functions such as *memmove*, *snprintf*, *getline*, and *posix_memalign*. We believe our work will benefit both the protocol developers as well as the users by taking the first step to ensure the security of IB protocol.

I. Introduction

The rapid growth of various applications such as social networking, mobile computing, and search engines has created large volumes of complex data. This requires a very real need to store, transfer, and process this high volume of unstructured data. Legacy network interfaces are not suitable for satisfying such strict requirements, as they all suffer from a major bottleneck – the operating system's kernel needs to make at lease one copy of the incoming/outgoing data (from the network interface) internally before making it available at the user level buffer [12]. The best solution for addressing this problem is the use of a remote direct memory access (RDMA) system [15], which provides the ability for one machine to directly access (read/write) the memory of another remote machine. RDMA has evolved from DMA which allows direct memory access in local machine without CPU involvement. There are different network protocols that support RDMA such as InfiniBand (IB) [7], RDMA over Converged Ethernet (RoCE) [7], and Internet wide area RDMA Protocol (iWARP). All of these network protocols sharing the same application programming interfaces (APIs) form a completely new networking protocol layer. IB was proposed by InfiniBand Trade Association (IBTA) as an open standard, interconnect protocol to overcome the bulk data transfer problem. It can provide the most efficient way for storing and processing large volumes of data which are typical in data center operations. In addition, it supports server virtualization, overlay networks, and software defined network (SDN), making it feasible as the inter-connect networking protocol for the foreseeable future. Compared to RoCE and iWARP, IB offers several advantages, including high bandwidth, low latency, good quality of service, great scalability/flexibility, high CPU offloads, and easy management.

Given that IB is quickly gaining in popularity, ensuring its security is imperative. A recent vulnerability in IB implementation is the use of *write* function as bi-directional *ioclt()* which is fundamentally unsafe, as it allows triggering *write* function calls from different contexts to carry out privilege escalation [4]. To ensure the security of IB, it is non-trivial to understand the security of IB implementations by performing a thorough security analysis, because: First, IB protocol is implemented by different vendors via hardware and device drivers. The diversity of the IB implementations may significantly increase the attack surface. Second, as a software component which is executed in the supervisor mode and provides interfaces for the kernel to access the hardware devices, device drivers may provide an avenue for attackers to escalate the privileges to the supervisor/admin level [9], since a substantial amount of device driver code resides in kernel space. It is thus very necessary to understand the security vulnerabilities of this code base.

However, it is challenging to perform security analysis on current IB implementations because: First, we lack specific tools for IB, as the available tools are unable to consider the vulnerable functions used in the IB implementations. Specifically, all the exiting open-source analysis tools which contain databases of the vulnerable functions are not updated with the most recent IB implementations. To confirm this, we tried to find all the functions used in the IB implementations using the existing tools, but we were not able to list all the functions being used as well as their corresponding frequencies of use. Second, the existing analysis tools do not consider the ratio between the number of the vulnerable functions and the lines of source code being analyzed, and we thus cannot obtain from these tools the different interfaces involved to accomplish the data transfer between two nodes. Third,

performing dynamic analysis on IB requires setting up new devices, which further requires installation of device drivers as well as setup of network components. However, it is usually complicate to ensure correct configuration and setup.

In this paper, we perform a comprehensive security analysis on the implementations of IB protocol from nine different vendors. The implementations include all the device driver application programming interfaces (APIs) as well as the user-level libraries. We start by performing a static code analysis on the IB implementations using various static analysis techniques and tools. To have a thorough understanding of the run-time behavior of the program, we also perform dynamic analysis by setting up and configuring necessary infrastructures. To the best of our knowledge, we are the first to evaluate the security of IB implementations from a system's purview. Our security analysis revealed that there are security vulnerabilities in the IB implementations: 1) The use of vulnerable functions like *memcpy*, *sprinf*, and *char*. The *memcpy* function does not check the length of user controlled input when copying data from source to destination. This may allow the attacker to overlap other region of memory with the code he/she controls. The secure alternative of *memcpy* function is *memmove*. Similarly, the secure alternative for *sprintf* and *char* are *snprintf* and *getline*, respectively. 2) The use of vulnerable functions in user-level libraries like *memalign*. The *memalign* function will return the block of memory in a successful case otherwise it will return NULL pointer. Its security impact is huge as the attacker can control the return value and redirect the control to his/her own choice of code. The *memalign* function can be replaced by either *posix_memalign* or *aligned_alloc* which will return zero upon being unable to allocate requested memory size.

II. BACKGROUND

A. IB Protocol and Architecture

The InfiniBand system mainly consists of three components: the user-level library, the device independent application programming Interfaces (APIs), and the device drivers. The user-level library runs at user level, while the device independent APIs (i.e., kernel verbs) and device drivers are executed at kernel level. InfiniBand is a new architecture to support higher bandwidth as well as to provide support on legacy architecture. A general architecture of IB software components is shown in Figure 1. Note that the most commonly available user level library is libibverbs. InfiniBand is a layer-based protocol as depicted in Figure 2. As a new interconnect architecture which supports different layers where each layer provides functionality to the upper layer. InfiniBand also supports services to the existing TCP/IP network interconnect. IB supports four different connection types at layer 2: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD), and Unreliable Datagram (UD). IB utilizes a queue-based model: a Send Queue (SQ), a Receive Queue (RQ), and a Completion Queue (CQ). The transport layer provides different level of queues, and each queue handles four types of data transfer including *Send/Receive*, *RDMA Write*, *RDMA Read*, and *RDMA Atomics*.

IB physical architecture. IB devices use serial links which are high-speed and bi-directional. It comes at different data rates such as single data rate (SDR) and double data rate (DDR).

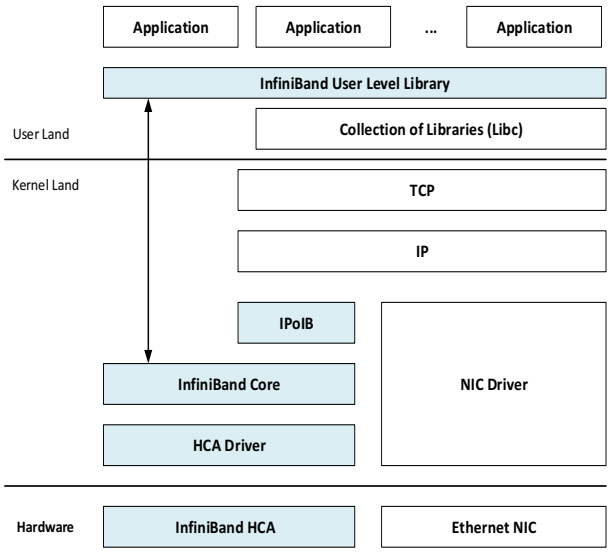

Fig. 1: General architecture of InfiniBand software components [5]

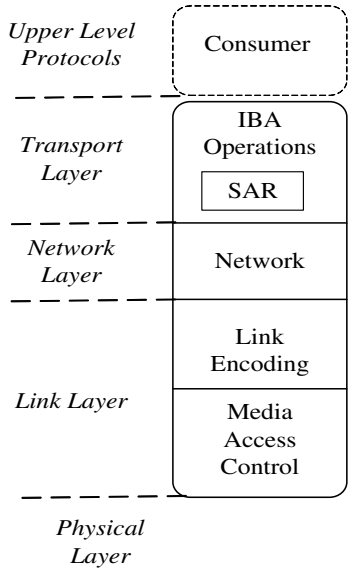

Fig. 2: Different layers of InfiniBand and their inter-dependency [18]

IB protocol packet formats. IB uses a routable packet unit that carries IBA operations in the fabric. The packet [22] is built using many different fields. The fields are Local Routing Header (LRH), Global Route Header (GRH), Base Transport Header (BTH), Extended Transport Header (ETH), Payload (with size 0 to 4096 bytes), Invariant Cyclic Redundancy Check (ICRC), and Variant Cyclic Redundancy Check (VCRC). This information is very important to build attack surface (CVE-2001-0144).

B. Taxonomy of Kernel Vulnerabilities

Our approach uses the following taxonomy of kernel vulnerabilities [21]:

1) **Nonvalidated / Uninitialized / Corrupted pointer dereference:** This is one of the famous kernel bug classes. This class is related to a pointer declared as a local variable in a function.
2) **Memory corruption vulnerability:** There are two types of kernel memory: the kernel stack, which is associated with each thread/process whenever it runs at the kernel level and kernel heap, which is used each time when a kernel needs to allocate a small object or some temporary space. Each user-land process runs on a system that has at least two stacks: user-land stack and kernel-land stack. The kernel stack is used when the process switches to kernel space by calling system function calls. The kernel stack is limited in size (4KB or 8KB on x86 architecture), and this is why the paradigm of using has few local variables as possible when preforming kernel programming. So, kernel stack vulnerabilities are not much different from their user-land counterparts and are usually the consequence of writing past the boundaries of a stack allocated buffer. The vulnerabilities occur due to the using of unsafe functions such as strcpy() or sprintf(). The stack plays a very important role in the application binary interfaces (ABIs) of a specific architecture, hence exploiting kernel stack vulnerabilities can be architecture dependent. Kernel uses a separate memory allocator, slab allocator, to allocate memory in kernel level. There was a vulnerability on slab allocator in old version of kernel, namely, CVE-2008-1673.
3) **Integer issues:** The most common type of bugs related with integer is: integer overflow and sign conversion issues. This class of vulnerabilities is not exploitable but it may lead to other vulnerabilities such as memory overflows. This is due to wild multiplication/increments/decrements. The sign conversion issues are related by evaluating some integer values as signed by integer first and then as an unsigned one (or vice versa).
4) **Race conditions:** Race condition is defined when tow or more processes are about to perform an operation and the results from their operation will be different depending on the order they execute. There are different synchronization mechanisms used in kernel such as locks, conditional variables, semaphores, etc. Race condition is very difficult to trigger, i.e., small trigger windows.
5) **Logic bugs:** There are three types of common logic bugs: kernel-generated user-land vulnerabilities, physical device input validation, and reference counter overflow. Kernel-generated user-land vulnerabilities are present in the interface between kernel and user space such as udevd. Physical device input validation vulnerabilities are becoming popular because device drivers are not able to handle a system crash. For example, there is a recent usb attack on Windows 8 and MAC OSX login by pass [20]. Reference counter overflow vulnerabilities are common while keeping track of the resources.

C. Tools Used in Our Security Analysis

We rely on three tools to perform static analysis: *Flawfinder*, *Splint*, and *Sparse*. The dynamic testing is performed using *Melkor*.

1) **Flawfinder:** Flawfinder [27] is a static code analysis tool which has a database of C/C++ functions with well-known issues, such as format string vulnerabilities ([v]snprintf(), [v][f]printf(), and syslog()), buffer overflow vulnerabilities (strcpy(), gets(), sprintf(), scanf(), and gethostbyname()), race conditions (access(), chown(), chmod(), tmpfile(), tmpnam(), tempnam(), and mktemp()), potential shell metacharacter dangers (exec() family, system(), popen()), and poor random number generator (random()).
2) **Splint:** Splint [13] is one of the most popular open source tools for analyzing source code statically. We run Splint against all of the kernel modules, and use default annotation provided by Splint.
3) **Sparse:** Sparse [25] is a semantic parser which can provide front-end capability of parsing most of GCC extensions as well as ANSI C program. We use Sparse [25] to analyze the IB component of the source code while compiling the kernel source code using *$ make C=2 staging/drivers/infiniband/*.
4) **Melkor:** We use Melkor [16] to generate test data to perform fuzz testing.

III. APPROACH

Performing security analysis on the software written in native languages such as C/C++ is necessary to find out the security weaknesses while implementing the given specifications of different vendors. We perform static analysis on IB implementations using our own developing static analysis tools to enumerate all the possible vulnerable functions in the application programming interfaces (APIs), and using the existing static tools to carry out rigorous analysis such as null pointer dereference, type mismatching, etc. After having performed static analysis to find out lower hanging weaknesses, we also perform dynamic analysis to find out the functions and parameters being used when the system is in running state. In the following, we first outline the rationale behind our approach, and then describe the details of our static and dynamic analysis on IB.

A. Rationale

To understand the vulnerabilities of IB implementations, we perform both the static analysis and the dynamic analysis on them. The static analysis is performed using our static analysis tool which is built specifically for IB implementations. The code bases from nine different vendors are processed, detecting the suspected list of vulnerable functions with corresponding frequencies. We use code annotation to detect vulnerabilities in functions such as *null parameter*, *tainted data*, *control flow analysis*, and *secure coding practice*. We perform code analysis by doing code annotation to test for nonvalidated, uninitialized, corrupted pointer dereference, memory corruption vulnerability, etc. In this way, we can find out any missing security related issues in the IB implementations.

Only applying static analysis is usually not enough as it is not able to capture the execution behavior of each software component. As we know, *the security of a system is only as strong as its weakest link* [14]. The system's execution behavior shows its actual flow of data through the interfaces. By enumerating all the interfaces involved as well as the data flow, we are able to obtain the weakest links if present. One of the common problems in security is *user controlled input data*. When the user controlled input data violate the data context and are interpreted as code, a "code injection" attack happens.

The IB consists of different interfaces which are including from build-in libraries as well as the IB application programming interfaces. These interfaces are very important to reveal any security issues. We perform dynamic analysis to capture the execution path and the data flow among all the APIs. The dynamic analysis requires setting up and configuring hardware devices specific to IB and all the software components involved. We run each user level program and capture executions of the programs specific to IB using **ltrace** and **strace**. We also develop a tool which can process the output of these functions and create the state diagram. This tool is very flexible and can be used by other tools to create a state-diagram with output. In addition, We perform fuzz testing on each module in kernel space. Fuzz testing covers each section of executable and linking format (ELF). The purpose of fuzz test is to cover the input validation while parsing ELF files.

We also perform dynamic analysis on the IB components during running. The dynamic analysis provides all the application programming interfaces involved while moving data from source to destination. We try to transfer bulk data to learn how the system handles large-size data. This is one form of fuzz testing in dynamic analysis.

B. Static Analysis

Static analysis is a technique which can find out vulnerabilities in the source code without running it. In general, static analysis requires the source code of the software being analyzed. For IB implementations, we can have access to the source code. Static analysis can be performed using two options: manual code review and automatic scanning tools. Manual code review however, does not scale well for large code bases, and thus we rely on automatic tools.

In our static analysis, we first use tools to gather statistics on the source code, and then use the static analyzer to find out vulnerable functions. Once we obtain the name of the vulnerable functions, we further use white box testing to detect vulnerabilities. Note that these vulnerable functions will be the primary target to perform dynamic analysis.

1) Gathering the Source Code Statistics: We use the static analysis tool we implemented to obtain statistics on the source code including the number of c program files, header files, line of source code, number of structures. The results of our purposed static analyzer will be presented in Section IV.

2) Detecting Vulnerability: Once having gathered necessary information on the target source code, we perform vulnerabilities discovery using white box testing to perform static analysis, which is accomplished either with the assistance of automation tools or manually. We use our developed static analyzer to perform initial static analysis to find the number of vulnerable functions with respect to the size of source code, and then use the automated tools (Flawfinder, Sparse, and Splint) to perform white box testing. The automated tools fall into three categories - compile time checkers, source code browsers, and automated source code auditing tools [24]. The source code is analyzed using Splint. We use following example code (see the sample code below) snipped to find the vulnerabilities.

```
Bisection($/*@only@*/ char *link$)
    ...
    return $*link$
Demo1($char *s$)
    return $*s$
Demo1_with_annot($/*@null@*/ char *s$)
    return $*s$
```

C. Dynamic Analysis

Static analysis is unfortunately not able to capture runtime behavior of the program (as mentioned in [24], "*Keep in mind, however, that what you see is not necessarily what you execute when it comes to source code*"). We thus perform dynamic analysis to address this limitation. The dynamic analysis is usually performed when the target application is running, and can provide the exact flow of data from different interfaces involved. Specifically, it can capture the exact run of the programs [8] and can help detect vulnerabilities which cannot be detected by static analysis. For example, by using dynamic analysis, a recent vulnerability is found in glibc library function [2].

The dynamic analysis is performed using fuzz testing. When using fuzz testing, we consider only the input and output relation. We generate crafted inputs and send these inputs, and observe the response of the running program. The main advantages of fuzz testing are availability, reproducibility, and simplicity, and the main disadvantages are code coverage and intelligence. The testing is performed in the executable file formats, which have different sections such as data section, code section, stack section, etc. ELF (Executable and Linkable Format) [11] is a common file format for executables, object code, shared libraries, and core dumps in Linux-like operating system. The working ELF modules from nine different vendors are used as a seed. We perform fuzz testing on all the modules used in IB from those vendors. *Melkor* is used to generate fuzz data, by which we generate five thousand test data to fuzz each module. We observe kernel logs in monitor mode while loading and unloading each module.

At a high level, there are three steps to perform dynamic analysis: *installation and configuration*, *monitoring execution flow*, and *capturing the execution flow*. To have a clear picture from the captured execution flow, we also create a state diagram by processing the execution flow which can show the functions being used and parameters being passed from one function to another.

IV. EXPERIMENTAL RESULTS

In this section, we present our experimental results for our static and dynamic analysis.

TABLE I: IB source code statistics: number of C programs, functions, data structures

Count	2.6.18		4.2.3	
	INCLUDE/RDMA	DRIVERS/INFINIBAND	INCLUDE/RDMA	DRIVERS/INFINIBAND
C FILES	0	76	0	221
HEADER FILES	16	26	20	110
LOC	5411	17185	7388	27410
STRUCT	704	236	996	1063
ENUM	99	62	180	379
UNION	28	2	49	55
MEMCPY	4	120	10	478
MALLOC	0	0	0	0
KMALLOC	0	140	0	318
KFREE	0	371	0	1275
REALLOC	0	0	0	0
DELETE	0	0	0	0
GOTO	0	1352	0	4246

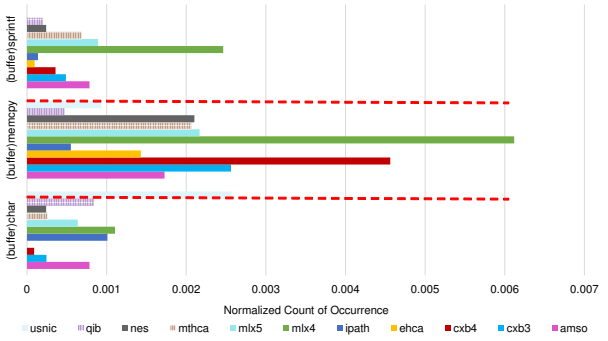

Fig. 3: Flawfinder results of Linux kernel 2.6.18

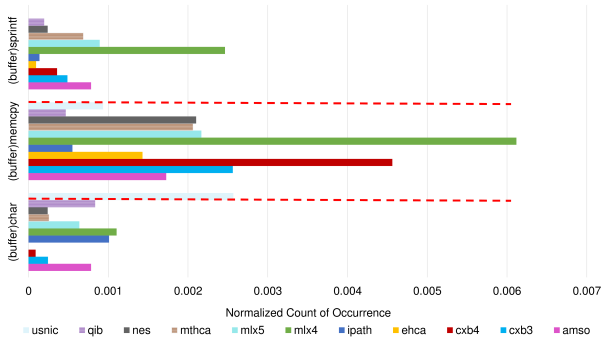

Fig. 4: Flawfinder results of Linux kernel 4.2.3

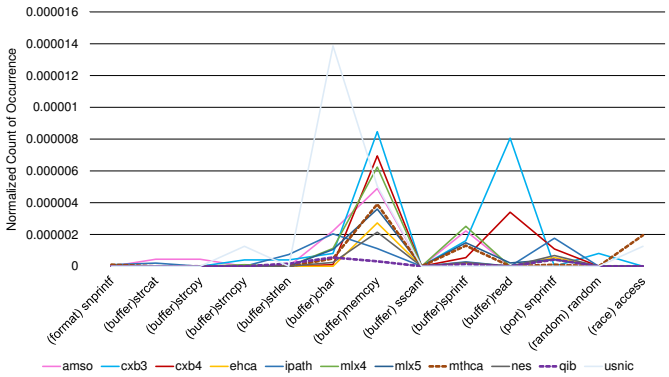

Fig. 5: Comparison between Linux kernel 2.6.18 and 4.2.3

Experimental setup. We used two servers with hardware and software components (server 1: AMD Opteron 4256 EE 1.6GHz, Linux 2.6.18-406.el5; server 2: Intel Xeon X5650 @ 2.67GHz, Linux 2.6.18-406.el5). We used Mellanox Certified Refurbished MC1104130-001 Passive Copper Cable InfiniBand CX4, which supports DDR 20Gb/s. We also used the Infiniband adaptor QLE 7240.

A. Static Analysis

We used our purposed static analyzer to perform initial code analysis, and the results are shown in Table I. We then use Flawfinder, Splint, and Sparse to perform static analysis and show the results respectively.

1) Flawfinder Result: We ran Flawfinder over the source code in the system where we have installed hardware device. According to the QLogic recommendation, we used CentOS 2.6.18-406.el5 with Linux kernel 2.6.18. Flawfinder discovered a number of vulnerabilities based on the taxonomy described in Section II-B. As we know, the vulnerability counts vary with the lines of source code (LOC). We thus normalized these counts using Formula 1.

$$Normalized\_Count = \frac{counts}{\#LOC} \quad (1)$$

All the results are shown after the counts are normalized. Figure 3 shows three important functions which cause vulnerabilities if used incorrectly: *(buffer)char*, *(buffer)memcpy*, and *(buffer)sprintf*. These functions are vulnerable to memory overwrite, memory over-read, and format string. Different colors show different vendors and bar lines show the frequency of these functions. In particular for each outlier, we observed that the mlx4 driver has the highest number of vulnerabilities. In addition, we observed that (buffer)memcpy is more prevalent than the other outliers.

Since the IB implementations in kernel 2.6.18 is comparatively old compared to the latest stable kernel 4.2.3 (at the time of our evaluation), we thus ran Flawfinder using kernel 4.2.3 and plotted those three outliers: (buffer)char, (buffer)memcpy, and (buffer)sprintf. Note that we also used the normalized counts of vulnerabilities. The results are shown in Figure 4. We observed that the plot pattern in both versions (2.6.18 and 4.2.3) is similar. To further evaluate the differences, we also plotted the absolute difference of the normalized counts of the static code vulnerabilities, and the results are shown in Figure 5. We observed that in the latest IB, the aforementioned three vulnerabilities have been fixed in a certain extent. However, they still exist (Figure 4).

2) Splint Result: We performed data-flow and control-flow analysis using *Splint*. Spring has the ability to interpret special annotations to the source code, and can be used for various tests, including null pointer, memory leaking, dangling pointer, buffer overflow, etc. There were no serious vulnerabilities being found in Splint.

3) Sparse Result: We performed static type-checking using Sparse. Sparse is used while configuring the kernel to enable IB kernel drivers. There were no serious vulnerabilities being found in kernel drivers.

TABLE II: Kernel level ELF modules

Module Name	Type	Size in Memory (bytes)
amso1100/iw_c2.ko	ELF 64-bit relocatable, not stripped	Module Error
cxgb3/iw_cxgb3.ko	ELF 64-bit relocatable, not stripped	109725
cxgb4/iw_cxgb4.ko	ELF 64-bit relocatable, not stripped	Module Error
ipath/ib_ipath.ko	ELF 64-bit relocatable, not stripped	361893
mlx4/mlx4_ib.ko	ELF 64-bit relocatable, not stripped	361893
mthca/ib_mthca.ko	ELF 64-bit relocatable, not stripped	156644
nes/iw_nes.ko	ELF 64-bit relocatable, not stripped	223209
qib/ib_qib.ko	ELF 64-bit relocatable, not stripped	Module Error

B. Dynamic Analysis

We configured and set up two servers to perform dynamic analysis, using QLE 7240 IB adaptors. The necessary condition for the dynamic analysis is a properly running environment. The dynamic analysis is performed using fuzz testing and the run time execution is monitored. The fuzzing testing is carried out on all the available kernel modules of all vendors, and the run time execution is monitored using *ltrace* and *strace*, which are used to find all the system calls associated with user level programs [23]. The corresponding state diagram was drawn based on the output of ltrace and strace.

1) Fuzz Testing: We performed fuzz testing, a technique closely related to boundary value analysis (BVA) [19], to obtain better coverage of the security analysis over IB. The ELF header contains meta-information specific to particular file, and we fuzzed headers of all the modules possessed by the user space ELFs using Melkor Fuzzer. We used the user level ELFs as our seed input. We generated six sets of test data using different values of n (i.e., 500, 1000, 1500, 2000, 2500, and 3000) and the following command snippet: # ./melkor -l 20 -n {500, 1000, 1500, 2000, 2500, 3000} templates/elf_name.

The "templates" directory contains all the modules, and we used each module as test data to generate fuzzed data. All these data sets were used to check whether all kernel modules can crash or produce errors. We did not find any such crash or errors while running executables. We also observed that all the user-level ELFs are stripped which is good from the security perspective.

Similarly, we performed ELF section fuzzing of all the kernel modules (Table II). After having generated all the test kernel modules, we loaded each module. As shown in Table II, three of the vendor's modules crashed. We monitored the kernel log to find the reason behind the crash, but unfortunately, we have not identified any significant reasons for the crash. We also observed that all the kernel ELFs are not stripped. We recommend that all the modules should be stripped for security consideration. In addition, five other vendors' modules were loaded successfully without causing errors.

2) API Call Graph: The dynamic analysis is performed using *strace* and *ltrace*. The output produced by both tools were further processed using customized Python program to generate **TikZ** file, by matching function names specific to IB operations. Then, we used the **TikZ** file to draw a state diagram in Figure 6. The diagram offers a clear picture about which functions are used while IB protocol is in used. The node of the state diagram represents the function name and edge represents the order in which these functions are called.

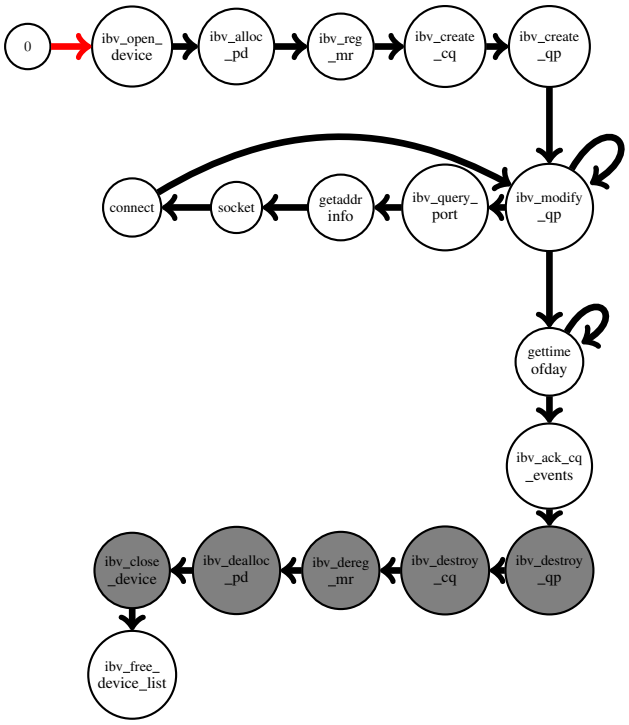

Fig. 6: State diagram of *ibv_uc_pingpong*

V. RELATED WORK

Aron Warren [26] performed an equivalent of Ethernet's MAC address spoofing vulnerability in InfiniBand. The equivalent vulnerability in IB is an HCA GUIDs poisoning attack. Manhee *et al.* [17] considered several aspects of security such as availability, confidentiality, and authentication, but they only mentioned enhancing confidentiality by using Invariant Cyclic Redundancy Check (ICRC) field as an authentication tag.

The static analysis is one of the popular methods used to find vulnerabilities in software. There are a number of static analysis tools, including Flawfinder [27], Splint [13], and Sparse [25]. As we can see from the CERN [6], Flawfinder [27] is recommended because it is an effective static code analyzer for analyzing software written in C/C++. In addition to these open source tools, there are also several proprietary tools available like Coverity [1], Fortify [3], and MAYHEM [10]. These tools can support more advanced analysis.

VI. CONCLUSION

In this paper, we report the results of our security analysis on InfiniBand, in particular, the overall implemented codebase [7] and the architecture. We use the penetration testing technique to discover vulnerabilities. To perform static analysis, we use Flawfinder, Splint, and Sparse. The black box testing is performed using Melkor. We checked on the parsing phase of user space and kernel space modules. We also gathered information regarding system calls being used and parameters of these system calls. We found four vulnerable functions, three are located in the kernel space and one is in the user-space libraries.

VII. Acknowledgment

Authors would like to acknowledge supports from Center for Information Assurance (CfIA), University of Memphis.

References

[1] Coverity development testing. http://www.coverity.com/.

[2] Cve-2015-7547: glibc getaddrinfo stack-based buffer overflow. https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html.

[3] Hp fortify. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812.

[4] Ib/security: Restrict use of the write() interface. https://lkml.org/lkml/2016/6/12/209.

[5] Infiniband software and protocols enable seamless off-the-shelf applications deployment. http://www.mellanox.com/pdf/whitepapers/WP_2007_IB_Software_and_Protocols.pdf.

[6] Static code analysis tools. https://security.web.cern.ch/security/recommendations/en/code_tools.shtml.

[7] I. T. Association et al. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[8] T. Ball. The concept of dynamic analysis. In *Software EngineeringESEC/FSE99*, pages 216–234. Springer, 1999.

[9] S. Boyd-Wickizer and N. Zeldovich. Tolerating malicious device drivers in linux. In *USENIX Annual Technical Conference*, 2010.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 380–394. IEEE, 2012.

[11] T. I. S. Committee et al. Executable and linking format (elf) specification, 1995.

[12] U. Drepper. The need for asynchronous, zero-copy network i/o. In *Ottawa Linux Symposium*, pages 247–260. Citeseer, 2006.

[13] D. Evans. Splint-secure programming lint. Technical report, Technical report, University of Virginia, 2002. http://www.splint.org, 2002.

[14] N. Ferguson, B. Schneier, and T. Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011.

[15] K. Gildea, R. Govindaraju, D. Grice, P. Hochschild, and F. C. Chang. Remote direct memory access system and method, August 2004. US Patent App. 10/929,943.

[16] A. Hernandez. Elf parsing bugs by example with melkor fuzzer. http://www.ioactive.com/pdfs/IOActive_ELF_Parsing_with_Melkor.pdf.

[17] M. Lee, E. J. Kim, and M. Yousif. Security enhancement in infiniband architecture. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 105a–105a. IEEE, 2005.

[18] Mellanox. Introduction to infiniband, 2012.

[19] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services*. University of Wisconsin-Madison, Computer Sciences Department, 1995.

[20] S. Mushell. Bypassing windows and osx logins with nethunter & kon-boot. https://www.offensive-security.com/kali-linux/bypassing-windows-and-osx-logins-with-nethunter-kon-boot/.

[21] E. Perla and M. Oldani. *A guide to kernel exploitation: attacking the core*. Elsevier, 2010.

[22] G. F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[23] D. Stolz and A. Grest. Trace collection, analysis, and visualization for barrelfish. *Distributed systems lab, ETH Zurich*, 2013.

[24] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[25] L. Torvalds, J. Triplett, and C. Li. Sparse a semantic parser for c. http://sparse.wiki.kernel.org.

[26] A. Warren. Infiniband primer and basic fabric attacks, 2012.

[27] D. A. Wheeler. Flawfinder, 2011.