Securing Knowledge Queries Using Code Striping

Mark W. Bailey
Computer Science Department
Hamilton College
Clinton, NY 13323
mbailey@hamilton.edu

Kevin Kwiat
Air Force Research Laboratory
525 Brooks Road
Rome, NY 13441-4505
kwiatk@rl.af.mil

Keywords: secure code, remote execution, redundancy, fault-tolerant system, distributed voting.

Abstract

Remote execution of programs raises security concerns for both server that executes the program and the program itself. We propose a solution, called code striping. Striping uniquely provides simultaneous protection of the server from the client query and the client query from the server using a single mechanism. By combining striping and distributed voting, we provide protection for code not currently available and a higher level of protection to the server than currently possible using present methods.

1 Introduction

As the Internet has grown in the last decade, so has the demand for remote execution of programs. When a server maintains a knowledge base, a client may transmit code, or a query, across the network to a server that subsequently executes the code. We must protect the server from malicious code it receives from the network, and we must assure the client that the result it receives from the server is the result of executing the client's program. In this paper, we propose a technique for protecting code from the host as well as protecting the host from the code.

A client/server model of computation, where the client provides a query for the server to perform, is often used when it is more efficient for the client to transmit a query to the server than it would be for the server to transmit query data to the client (due to bandwidth limitations, response time requirements, or security considerations). Figure 1 shows the architecture of such a client/server fault-tolerant distributed system. In situations where fault-tolerance is critical, multiple, redundant servers may be used, where each performs the same query and communicates its results to a poller. The poller compares the results and masks erroneous server results by communicating only the majority result to the client.

2 Background

Erroneous query results may occur for a variety of reasons, including: interception and modification of a client query en route to a server, or compromise of a server by an outside agent. For such a system to be effective, the client and servers must agree to cooperate. The client must trust that the servers will perform faithfully the client's query and the servers must trust that the client's code will not attack the servers.

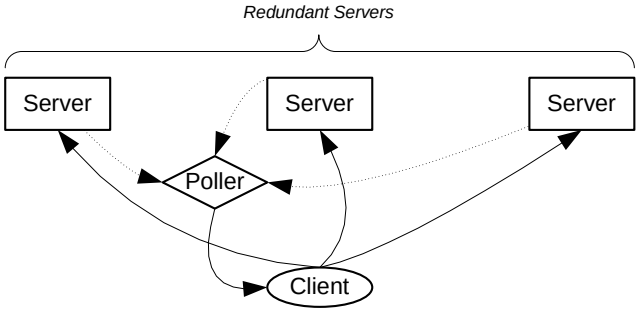

Fig. 1 A fault-tolerant distributed system with redundant servers and a poller.

We increase the security of queries and servers in the presence of outside attackers. The query is protected from modification and faulty execution in much the same way that data is protected in quality industrial data storage systems. In data storage systems such as RAID [2], data integrity is ensured using a combination of redundancy and distribution of data. We believe these techniques can be applied equally well to queries as to data with similar advantages.

RAID (Redundant Array of Independent Disks) storage systems maintain data integrity by using arrays of disks [2]. Although RAID uses an array of physical disks, it presents a single logical storage volume to the machine. A storage system using a block-interleaved distributed parity RAID (Level 5) configuration, shown in Figure 2, uses an array with N disks, where each logical storage block is divided

into $N - 1$ sub-blocks called stripes. Each of these stripes is placed on a distinct disk in the array. The N^{th} stripe stores the computed parity (bit-wise exclusive OR) of the other $N - 1$ stripes. Thus, RAID employs distribution in its use of multiple physical disks and redundancy in its use of parity. The data redundancy enables recovery from failure of any single disk. When a disk fails, it will contain either a data stripe or a parity stripe for a given logical disk block. A lost parity stripe can be recomputed using the original $N - 1$ data stripes, while a lost data stripe can be recovered by computing the parity of the $N - 2$ data stripes and the parity stripe. Distributing the data across N disks also improves performance of the logical disk since each of the physical disks can be accessed in parallel.

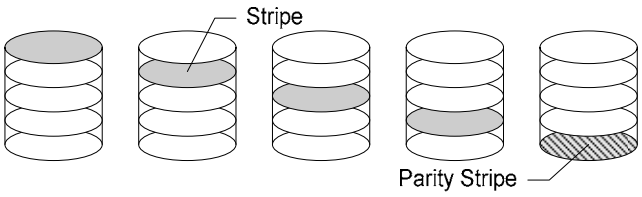

Fig. 2 Block-interleaved distributed parity (RAID level 5).

3 Code Striping

We couple RAID striping with a redundant, distributed system to yield a powerful mechanism for protecting the integrity of remotely executed code. Execution of remote queries is controlled at each server by the stripe virtual machine shown in Figure 3. The stripe virtual machine provides an environment in which queries execute and mechanisms for servers to start, stop, and resume execution of each query. This is achieved by two components: the *stripe execution environment*, and the *stripe state manager*. The stripe execution environment provides the server start, stop, and resume mechanisms, and any translation (through interpretation) between the query language and the native machine code. The stripe state manager enables the server CPU to capture the current memory state of a stopped query, transmit the captured query to other servers using the network adapter, and receive query state updates from other servers. Such an environment can be constructed using a Java virtual machine implementation [1, 7] or a dynamic translation system such as Strata [6], among others. Our code striping technique does not mandate any particular choice of implementation.

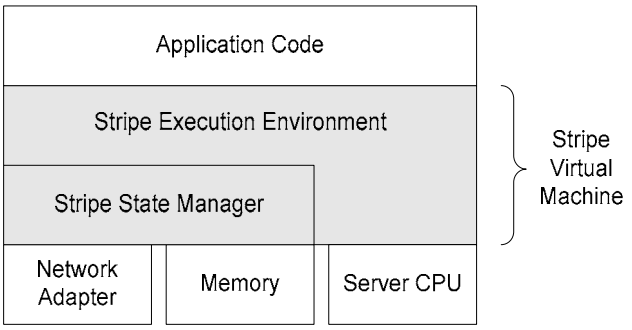

Fig. 3 Server stripe virtual machine.

Query execution is performed in parallel (for redundancy) where each query instance is distributed across all remote servers. Figure 4 illustrates how, conceptually, the execution of one query instance is distributed across the servers. First, the client transmits the query to the set of redundant servers. The servers divide identically the query into subtasks called *stripes*. The execution of each stripe is assigned to a remote server. The query begins execution on server *A*. When execution of the query reaches the end of the first stripe, the server captures the state of the execution. The server then transmits the state of execution to the server *B*, where the state is loaded into the stripe virtual machine and execution picks up where the previous server left off. This process of query execution followed by state capture and execution hand-off continues until the query completes execution on the server *D*. Server *D* then transmits the results of the query back to the client.

4 State Voting

The distribution of a query across a set of servers does not provide, in itself, any degree of fault tolerance. Without an additional mechanism, stripe execution may be corrupted by either a malicious intermediary or a compromised server. Figure 5 shows how an intermediary host could compromise a stripe. As the client transmits the query to each server, an intermediary intercepts and modifies a query en route to a server. In this case, the second server receives the modified query. Execution of the query continues correctly until the second server is reached. At this point, the modified query executes for the duration of the stripe. The result of executing the modified query is a corrupt state that is transmitted to the third and fourth servers, finally resulting in an erroneous result. An equally disastrous alternative occurs when a server is compromised. When control of the query execution is passed to the compromised server, the server either incorrectly executes the query, or simply modifies the query prior to execution. The result is the same: an incorrect final stripe state in the compromised server. This state is then passed through the

remaining servers where, again, an erroneous result is produced.

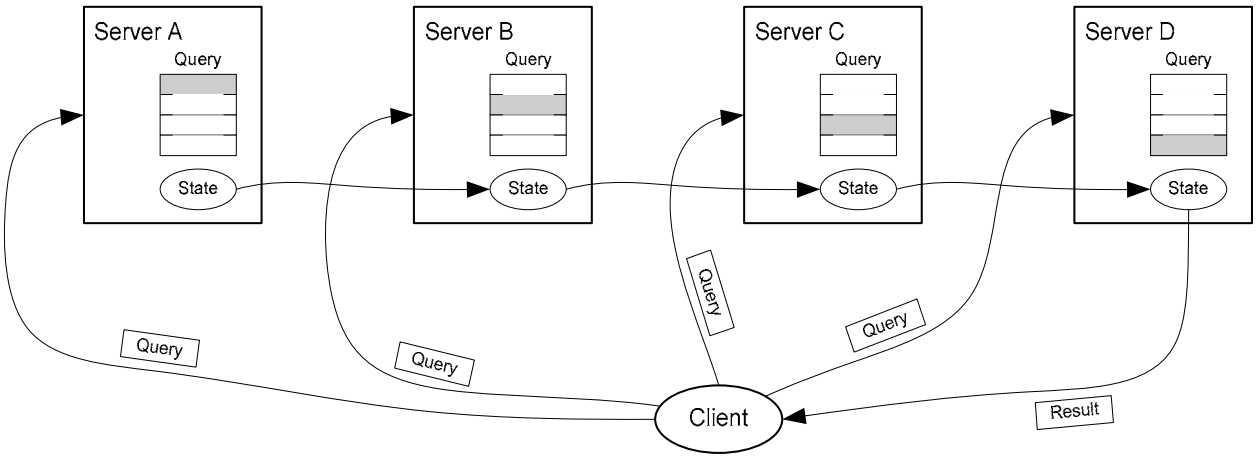

Fig. 4 Distributing query execution across servers.

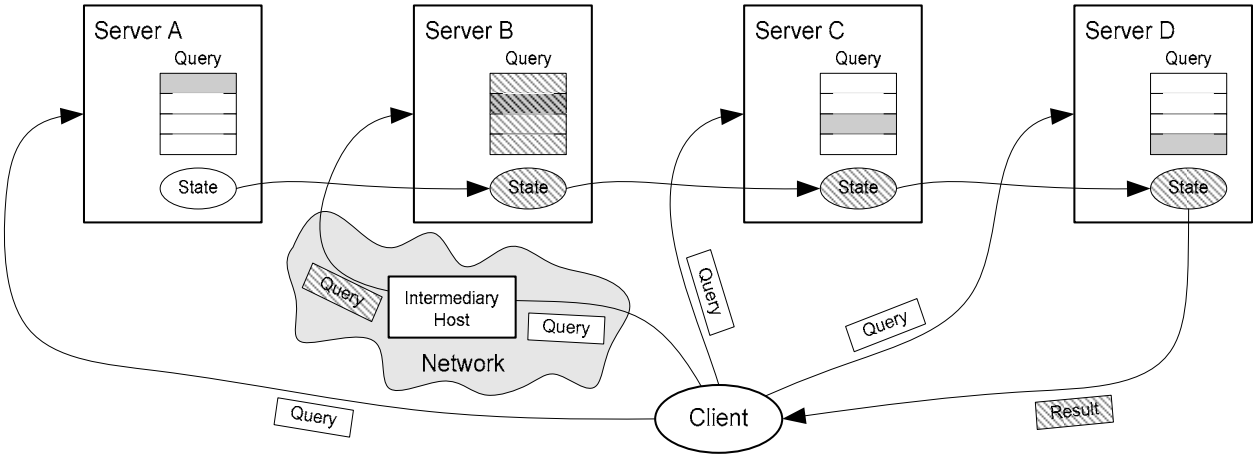

Fig. 5 Intermediary intervention of stripe execution.

Of importance to the client is the proper execution of its query. Should the client's query be modified by an intermediary along the way to a server, or the stripe be dispatched to a compromised server, the query might not be performed correctly. Code striping addresses these vulnerabilities by imposing the additional requirement that each stripe be executed on multiple (more than two) servers. This introduces the necessary degree of redundancy to recover from such situations. Now, upon completion of each stripe, each active server makes available to a poller the stripe's state of execution so it can be compared to the results from all other active servers. This comparison can be performed by a centralized poller or by a distributed polling algorithm by broadcasting the states (votes) to all servers [4, 3, 5]. The system determines the initial state of the next stripe by a majority vote of all participating servers. Thus, the degree of integrity of the system is determined by the number of redundant servers.

The stripe execution process that runs concurrently on each server provides protection for both the query and the server. Each stripe is executed on selected servers. Each server loads the stripe into the stripe virtual machine. The stripe virtual machine is initialized and executes the stripe. Upon completion of the stripe, the virtual machine halts execution and captures the state of the stripe's execution. The stripe's state is transmitted to the poller to determine the next stripe's beginning execution state. The poller forwards this new state to all participating servers, where it is loaded into the virtual machine. This process repeats until the entire client query is completed. At that point, the results of the query are voted on and transmitted to the client.

Figure 6 demonstrates how code striping annuls intermediary attacks, such as the one displayed in Figure 5.

Prior to the execution of stripe zero, server *C* receives and loads a modified query. During the execution of stripe zero, servers *A* and *B* perform identical queries while *C* performs the modified query. At the end of the stripe, all servers vote by forwarding their intermediate results to the poller. The majority result (that of *A* and *B*) is forwarded by the poller and loaded into all servers prior to the execution of the next stripe. This restores the state in server *C* to the correct state. The servers perform a second round of stripe execution. Again, *C* produces the incorrect result and *B* and *C* mask the erroneous result by voting it down. This process repeats, each time restoring the correct state by replacing the erroneous state. Finally, the last stripe's execution completes and the poller forwards the majority's state to the client. Although we allow the compromised query in *C* to continue, the redundancy and voting in the system ensures that an attack must modify a majority of queries in order to affect the client's result.

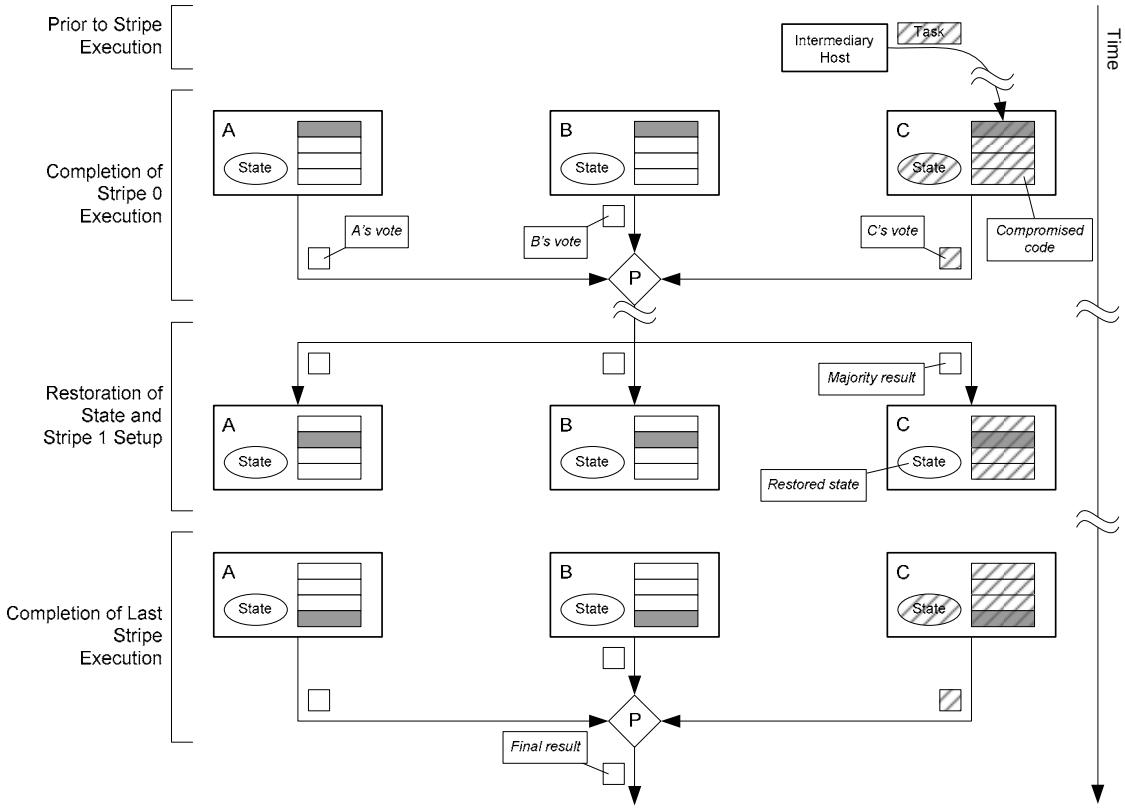

Fig. 6 Using striping and redundancy for secure knowledge query.

5 Protecting Code

In addition to intermediary attacks on client query, a compromised server can alter a client's query, or the execution of that query. Clients must be able to rely on servers to perform faithfully the client's requests. When a client's query is modified or incorrectly executed by the server, the result is the same as having an intermediary introduce an alternate query. Upon comparison by the poller, the compromised server's stripe state will be voted down by the majority. This effectively protects the client's query from attack by a compromised server.

Just as the client must trust the server to execute correctly the client's query, the server must trust the client query not to attack the server. The query striping mechanism can also protect the server from attacks from client code as shown in Figure 7. In this situation, an intermediary may alter a client's query as described previously, or it may introduce a new client query for an individual server to execute. Here, the goal of the intermediary is to compromise the server. Code striping protects the server by imposing a bound, *S*, on the stripe size (typically a constant). For any successful server attack, there is a lower bound, *M*, on the number of server machine instructions in the attack's query. Since we refresh each server's virtual machine state between stripes, no server preserves a minority's state from one stripe to the next. Since each server's virtual machine state is refreshed between stripe executions, no server preserves a minority's state is preserved from one stripe to the next. This places an

upper bound on the size of a successful attack of *S* machine instructions. Thus, a successful attack must be placed within a single stripe. Therefore, a stripe of size *S* will successfully thwart all attacks of size M > *S*. Consequently, as *S* is decreased, the level of protection is increased. In the extreme, S = 1, a successful attack must be encapsulated within a single instruction.

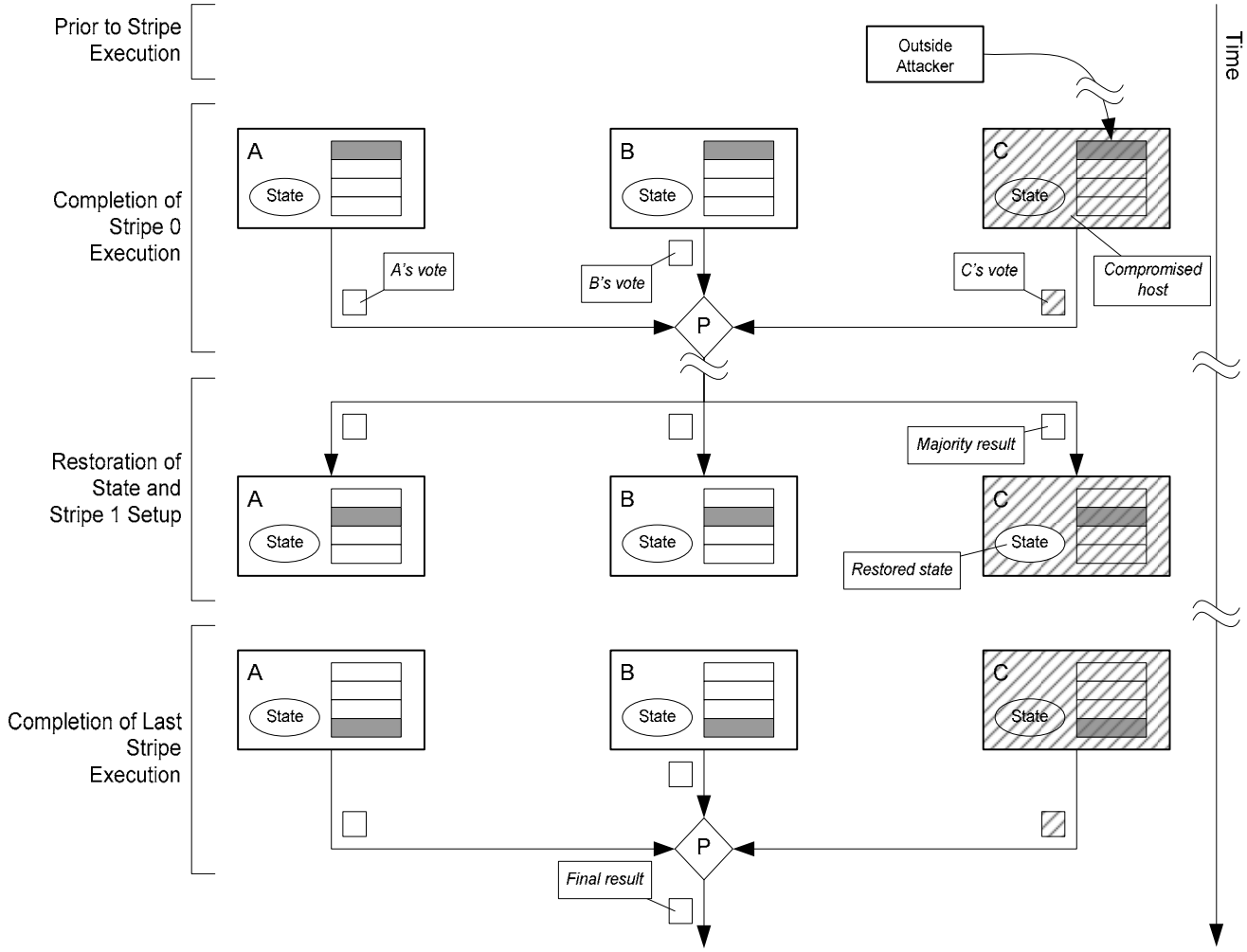

Fig. 7 Using striping to protect code.

6 Discussion

The use of striping does not mandate a particular stripe size, or a particular algorithm for determining stripe boundaries. The choice of stripe boundaries can impact the performance and security of the overall system. Since execution state must be transmitted and voted on at the end of each stripe, there is a significant performance advantage to choosing stripe boundaries that minimize the size—and thus the transmission time—of the execution state. A number of striping techniques could be used including: constant sized stripes, random sized stripes, partitioning outside innermost loops, partitioning at subprogram call sites, or partitioning at narrow points in the task's data flow graph. The only requirement is that each participating server must be able to determine the same stripe boundaries.

Although the choice of stripe boundary has significant impact on performance, the state transmission mechanism can also impact performance. In the previous description we assumed that the entire state is transmitted at each stripe boundary. An optimization can be performed in which only the state needed by the next stripe is voted on and transmitted to servers. We believe it is common for queries to build large data structures that then go unused for long execution periods. Such data structures do not need to be repeatedly transmitted. Another, similar, variant is to only

transmit and vote on changes to machine state. To be implemented efficiently, both methods require storage of machine state in a central location.

7 Summary

Client/server distributed systems commonly rely on remote execution of queries to improve performance, security, or both. Such an architecture relies on correct execution of client queries by servers. The addition of redundancy to improve fault tolerance makes these systems particularly prone to server and query attack. The benefit of using striping and execution state voting together is its holistic approach to providing protection to both queries and servers and its ability to configure the level of protection by varying the size of stripes.

Striping uniquely provides simultaneous protection of the server from the client query and the client query from the server using a single mechanism. By combining striping and state voting, we provide protection for code not currently available, and a higher level of protection to the server than currently possible using present methods.

References

1 M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, The Jalapeño Dynamic Optimizing Compiler for Java, In *proc. ACM 1999 conference on Java Grande*, 1999, pp. 129–141.

2 P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, RAID: High-performance, Reliable Secondary Storage, *ACM Computing Surveys* 26 (1994), no. 2, pp. 145–185.

3. B. Hardekopf, K.A. Kwiat, and S. Upadhyay. Secure and Fault-tolerant Voting in Distributed Systems. In *Proceedings of IEEE Aerospace Conference*, Big Sky (MT), March 2001, pp. 10–17.

4. B. Johnson, Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, 1989.

5. K. Ravindran, K. Kwiat, and A. Sabbir, Adapting Distributed Voting Algorithms for Secure Real-Time Embedded Systems. In *proc. 24th Intl. Conf. on Distributed Computing Systems Workshops*, Tokyo, Japan, IEEE, March 23, 2004.

6. K. Scott and J. Davidson, Safe Virtual Execution Using Software Dynamic Translation, *Proceedings of the 18th Annual Computer Security Applications Conference*, Dec 2002, pp. 56–61.

7 Sun Microsystems, *The Java Hotspot Virtual Machine*, v1.4.1, d2, Sep 2002.