Guide to Feralcore

Adam L. Young[1] and Moti M. Yung[2]

[1] Cryptovirology Labs
[2] Google and Columbia University

Abstract. We present feralcore, an experimental general-purpose communications protocol. Feralcore is a distributed communications protocol in which programs run in a shared-memory address space, can spawn new processes, and can send themselves anywhere within the feralcore network without any access controls. When a program is transmitted it carries with it a starting address. The receiving node unconditionally copies the program into memory starting at the specified address and executes it. This document explains briefly what the protocol is, how it is designed, how to deploy it, and an example application that we built on top of it. The application is a bulletin board protocol that allows users to post messages to a bulletin board. It is implemented using a feralcore program that atomically re-encrypts itself and retransmits itself to a randomly selected feralcore node. The extent of the anonymity provided is the subject of ongoing research. This is not a comprehensive research paper on feralcore. It is a brief introduction and a quick-start guide.

1 Introduction

Feralcore is a scientific experiment in network communications and cryptography. It is a self-contained network protocol, built on top of UDP, consisting of N nodes. These nodes form the complete graph on N vertices. It is a decentralized protocol in which nodes may join or leave the network. So, N may vary over time. Conceptually, each node is a virtual machine having an instruction set based on the Motorola 68000 (MC68000). Some Motorola instructions have been omitted since they do not make sense within the context of feralcore. The instruction set includes a feralcore API instruction that uses \$0008 as the operation word. This instruction allows programs to do complicated operations such as transmit themselves within the feralcore network.

A feralcore node is the Java program `feralcore.java` running in a Java Virtual Machine (JVM). It is the result of the JVM executing feralcore.main(). Feralcore is a large Java program that includes an implementation of a MC68000 virtual machine. When you compile `Feralcore.java` you compile this MC68000 virtual machine. So, when you run a MC68000 program in a feralcore node, you are running a program in the MC68000 virtual machine which in turn runs in the JVM.

To interact with a feralcore node, a user runs another Java program called `Fcdebug.java`. feralcore and fcdebug may or may not be on the same machine.

They use UDP to communicate with one another. fcdebug can be used to inject programs into the core, inspect memory, inspect the number of processes running in the core, and so on.

A *process* in feralcore is a set of feralcore registers. The feralcore registers are the Motorola 68000 registers plus a large register added to the virtual machine to support the feralcore protocol. A *program* in feralcore is a sequence of feralcore instructions. A process contains a program counter that points to the program being executed in the feralcore.

A feralcore node consists of a single memory core, a set of running processes, and a kernel that oversees the processes running in the core. In particular, the kernel manages the creation of processes, the execution of processes, the termination of processes, sending and receiving programs, and enabling programs to discover other feralcore nodes.

A process executes an instruction as follows. The kernel fetches the program counter (PC) from the process and then executes the instruction at that address in memory. The instruction may operate on memory as well as the registers of the process. Finally, the PC is updated for the process. For example, for the NOP instruction `0x4E71`, the value 2 is added to the PC. Since memory is shared, process A can interfere with the register values in process B by getting B to execute certain instructions.

The default configuration of feralcore is to use a core size of 16 megabytes. In this case the bytes start at address `0x00000000` and end at address `0x00FFFFFF`. Memory is handled in the same way that a real MC68000 chip would handle it. All 32-bit values are valid addresses. The most significant byte of a 32-bit address will be ignored by all instructions for this memory size. For example, reading the byte at address `0x30FFFFFF` is the same as reading the byte at address `0x00FFFFFF`. The Java code uses an interface class to implement core memory. A feralcore node could in theory be made to include all 2^{32} possible memory bytes.

Process Creation: A process can be created in several ways:

1. (kernel injection) The feralcore kernel has an injector class that injects programs into the core at random times. It randomly selects programs to inject from a set of predefined program types. This keeps things interesting.
2. (fcdebug injection) Using fcdebug users can inject their own programs into the core.
3. (network injection) When a program is received from the network the kernel copies it into memory and creates a process for it in the list of running processes. The program could be sent by another process in a remote node, or it could be sent from a user running a program designed to talk to feralcore nodes. In the former case, a process can send a program to a remote feralcore node by making a feralcore API call. The parent process selects the code that is sent. The parent could send a copy of its own program, or a different memory region in the core. This is a suicidal operation. The parent dies when transmitting the program. The child suffers a penalty delay upon arrival at the destination node. The delay is a counter that decrements each time the process would normally execute an instruction. When it reaches zero, the

kernel will begin executing intructions in the process. The registers of the process are not transmitted along with the program code.
4. (process spawning) A process can spawn a new process by calling a specific API call for this purpose. However, when this is done the parent and child suffer a penalty delay.

Process Execution: The feralcore kernel uses a randomized scheduler to execute programs. Suppose that M processes are executing in the core. Let the processes be labeled 1 through M. The kernel has a coin with M sides labeled 1 through M. The kernel flips the coin and notes the result, call it I. The kernel then executes a single instruction in process I. This is repeated ad infinitum, using coins that match the number of running processes. The kernel does things in between tosses such as accept new programs received from the network. This randomized approach was chosen over a round-robin strategy to simplify the code and make activities in the core less predictable.

Process Termination: A process can terminate in several ways:

1. (quota exceeded) When the process quota is exceeded the kernel selects a process randomly and terminates it. In more detail, a process is selected randomly from the list of running processes, is removed from the list, and then the new process is added to the list. For example, suppose that the quota has been reached and then a program is received from the network. Feralcore will select a random process and terminate it by removing it from the list of running processes. The kernel will then copy the code of the new program into memory and create a corresponding process for it in the list of running processes. As a result, the quota has still been reached, but some unfortunate process has been killed by the kernel.
2. (illegal instruction) A process that executes an illegal instruction is immediately terminated by the feralcore kernel. A perfectly normal way for a process to terminate itself is to execute an illegal instruction. So, when coding a program in assembly it is normal to get the program to terminate by making it execute an illegal instruction. Also, one way for one process to wipe out another is to get it to execute an illegal instruction.
3. (packet loss) UDP is a best-effort protocol. It does not guarantee delivery. The API calls that transmit programs are designed as follows. Each call sends out the supplied program multiple times to the destination feralcore node. But, there is no guaranteed delivery. A process can die if all of it's packets are lost during transmission.
4. (queue dropping) The OS that hosts the feralcore node receives processes as UDP packets. What happens if the queue of UDP packets gets too long? The OS might drop packets. A process can die due to UDP queue overflow.

This covers the lifetime of a process in feralcore, from creation to termination. From this we see that the basic operations that a process can perform are: spawn a new process, send a program to a remote node, and most of the MC68000 instructions.

Feralcore is based on a cryptovirology protocol outlined in Chapter 3 of [21]. Ideas related to feralcore include Darwin from Bell Labs [2], Core War [6], the Xerox PARC Worm [16], Avida [1], Network Tierra [5, 11, 12], and Active Networks [17, 18]. There are significant differences between feralcore and each of these notions and these differences will not be covered here.

2 Security

Measures were taken to ensure that feralcore prevents programs from escaping the core. Imagine the following threat. Suppose that feralcore were written in C++ and had a buffer overflow vulnerability and corresponding exploit. An attacker could craft a feralcore program that totes shellcode, send the program to the victim's machine, and have the feralcore program use the exploit. This would amount to a "real" program being transmitted through the feralcore network onto the victim's machine.

This threat is real and has motivated the measures we have taken to contain programs running in the core. To this end, we will now cover a bit about the history of our experiment.

2.1 Language Considerations in Terms of Security

An older version of the experiment was coded in C++ that used the MC6809 instead of the MC68000. The MC6809 is a hybrid 8/16-bit CPU. The C++ feralcore was over 10,000 lines of code. Despite relying heavily on C++ and the STL and following CERT secure C and C++ coding guidelines [15] it was abandonded for security reasons. We feared that C++ would not provide adequate security throughout the full software lifecycle of the protocol. This includes the inevitable modification of feralcore by however many programmers over time.

As a result, we redesigned the protocol and implemented it in Java from the ground up. Security was not part of the original design goals of C++. It was for Java. We did our best to follow the CERT secure Java coding practices in the current implementation.

The use of Java for feralcore has advantages over C++. The C++ feralcore was as POSIX compliant as we could make it. But, we still ran into compilation/link errors time and time again when we ported it to various platforms. The Write-Once-Run-Anywhere (WORA) dream of Java seems to be taking root and we have had amazingly few problems intalling feralcore on various systems.

Secondly, runtime-efficiency is not a primary goal of feralcore. Although the research is still ongoing, feralcore does not appear to be ideally suited for high performance computing (though this perception could change). It seems more suited for communication. In this regard, we can think of feralcore as a store-and-forward protocol. This lack of real-time performance constraints means that we can use a language that may not be as fast as C/C++. The feralcore protocol is pure middleware. It rides on top of UDP, is not a real-time protocol for the most part, and does not require a fancy front-end. Java excels when used as middleware. So, Java seems ideal at this time for feralcore.

When the Java version was off the ground, we gained a new perspective on the old version. Coding and using the C++ feralcore felt like performing a high-flying trapeze act without a net below. Sure, you might be confident your ability to grab the bar when it swings up to you. But, the guy standing next to you on the platform is not paying attention to his surroundings and is fidgeting for no reason. He may just knock you to the ground below. Java feels inherently safer for this sort of thing. Java always checks array indices, has no pointers, has portable and unambiguous primitive data types, inherently throws exceptions for a great number of exceptional conditions, garbage collects unreferenced objects, and auto-initializes local variables. No language is perfect, but we feel that Java is the best language for feralcore at this time.

It is important to design types that closely model the problem being solved. Failure to do so could lead to code that is harder to read, bugs, or even security vulnerabilities. We encountered a very subtle bug in an older version of the C++ feralcore that was due to the subtle integral promotion rules of ANSI C. The code below illustrates the issue. It prints out "!=". Would that be obvious to most code reviewers?

```
uint16_t x,y;

x = 0xFFFF;
y = 0x0000;
if (x == y - 1)
   cout << "==";
else
   cout << "!=";
```

As a result of this, we concluded that `uint16_t` was not up to the task of modeling a MC6809 16-bit register.

The operator overloading capability in C++ allows programmers to produce user-defined types that look, feel, and behave like built-in types. We leveraged this in the C++ feralcore to produce unsigned data types that directly model the MC6809. We created our own type "reg16" that models Galois Field (GF) arithmetic, i.e., operations modulo 2^{16}. This side-steps integral promotions since this is our own class. We used operator overloading, so reg16 looked exactly like the built-in C types when used.

Java does not provide operator overloading. Also, with the exception of type **char**, Java does not have any native unsigned types. It may have been possible to use the primitive Java twos-complement data types to model the MC68000. However, given the large amount of code used to implement the virtual machine this seemed inherently risky. The risk of introducing a sign related bug seemed too high.

So, in the Java feralcore we were left with the problem of producing data types to mimic the 68000 internally without being able to overload operators. We created our own unsigned types: reg2, reg3, reg4, reg5, reg8, reg16, and reg32. These implement unsigned integer types having the indicated length. For example, a reg32 is an unsigned 32-bit integer. The smaller types are used for modeling the smaller integers that appear as substrings of opwords.

This is the one area that the C++ feralcore outshines the Java feralcore. The C++ feralcore GF types produce simpler code than the Java feralcore GF types.

2.2 Static Code Analysis

We used three different static analyzers on our Java codebase: PMD, CodePro, and FindBugs. All three found unique things that we fixed. We used the PMD plug-in for Eclipse Indigo and the CodePro plug-in for Eclipse Indigo. FindBugs is written in Java and we ran it as a stand-alone Java application.

CodePro lets you audit the code using rule sets. We ran it with the CodePro rule sets, the Effective Java rule set that is based on [4], and the Security rule set. It is worth noting that collectively these static analyzers revealed the following things:

1. References that could be made final, including local variables, class data members, and arguments to methods.
2. Imports that are not used or needed.
3. Member functions that were declared to throw exceptions of certain types when these exceptions could not possibly be thrown.
4. Class data members that could instead be local variables.

2.3 Multi-Threading Considerations in Terms of Security

Even with a good language and careful coding practices, subtle security holes can creep into the implementation. Another large class of issues is race-conditions. This was such a concern that we decided to make the feralcore kernel single-threaded. It performs all of its tasks serially. It is conceptually helpful to think of the implementation as a “batch processing system”. Randomness is used throughout the implementation, so the ordering of certain tasks in the batch is randomized. This design makes it easy to reason about the operation of feralcore and facilitates a security code review.

The fcdebug program is multi-threaded only because it uses Java Swing to implement the GUI. We introduce no threads other than what Swing does natively. Fcdebug was specifically separated out from the feralcore process to help isolate the functionality needed to implement the core feralcore protocol. The feralcore process does not use Swing at all.

Race-conditions are not eliminated by having the code be single-threaded. Feralcore sends and receives packets and reads and writes files. So, there are still plenty of things to check when conducting a code review.

2.4 Feralcore Fuzzer

Feralcore has a built-in fuzzer. It is implemented using one of the programs that the injector injects into the feralcore. This program is a sequence of 1024 randomly chosen bytes. It is inserted at a randomly selected memory address subject to the following constraint: the address must be even. The reason for this is that the MC68000 can only access memory on even addresses and therefore all programs must run on word boundaries. By injecting this program, feralcore fuzzes itself regularly to try to identify bugs that may exist in the implementation.

2.5 Feralcore Crypto

A hybrid cryptosystem is used to encrypt the traffic between feralcore nodes. It uses 4096-bit ElGamal [7] and 3-key Blowfish [14] in cipher-block-chaining mode. Blowfish is used with three 128-bit keys and enciphering is performed using Encryption-Decryption-Encryption (EDE). The 4096-bit prime modulus p for ElGamal is a safe prime. That is, $q = (p-1)/2$ is prime. Also, $p \equiv 3 \bmod 4$. The ElGamal base g has order q. ElGamal is used in a way that conforms to [19] in order to achieve message security. However, the ElGamal implementation is further modified to achieve ciphertexts that appear as random octet streams. In particular, each byte in a ciphertext appears to all efficient algorithms as a string selected uniformly at random from $\{0,1\}^8$.

This randomization is achieved by: (1) making the 16 most signficant bytes of p be `0xFFFFFF...FF`, and (2) flipping coins to randomize the ciphertext values to be residues or non-residues with equal probability. This post-processing of the ElGamal ciphertext is computed using the non-residue -1. The fact that -1 is a non-residue follows from Euler's Criterion and $p \equiv 3 \bmod 4$.

These types of transformations on asymmetric ciphertexts date back to the probabilistic bias removal method [20]. See also [8, 3, 10]. The end result: feralcore packets look like random octet streams.

Feralcore supplements the Java `SecureRandom` RNG using the ISAAC pseudorandom number generator [9]. ISAAC has a massive internal state. We say *supplement* in the following sense. Our additional PRNG is combined with the native PRNG using the bitwise XOR operation (Santha-Vazirani extractor [13]). Finally, feralcore is configured to add new entropy to the PRNG state over time. In systems that provide it, this is done by reading `/dev/random`.

3 Node Discovery

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THIS SECTION IS OUT OF DATE, SEE THE PAPER *******

***** THE STRATEGIES HAVE CHANGED *********************

Feralcore would not be very useful if feralcore programs had no way of discovering other feralcore nodes on the network. A node discovery service is built into feralcore by way of the feralcore API. Feralcore programs can discover other nodes by making feralcore API calls. The feralcore kernel constantly learns about the current status of other feralcore nodes in the background. This is the decentralized peer-to-peer nature of feralcore.

An *fc3tuple* is a serverinfo object, public key, and timestamp. An fc3tuple corresponds to a particular feralcore node. The serverinfo object contains the network address and port of this feralcore node. It is therefore the full identify of a feralcore node. When a program on bob's feralcore node sends itself to alice's feralcore node, it gets encrypted with the public key in alice's fc3tuple prior to transmission.

The mutable class Fc3tuplecontainer.java stores the fc3tuples in a feralcore node. The maximum number of fc3tuples that is stored is dictated by the integer constant Feralcoreconstants.max_fc3tuples. If upon adding an fc3tuple to the fc3tuplecontainer the container overflows, then the oldest fc3tuple is deleted. The constructor of fc3tuplecontainer reads in fc3tuples that have been stored previously. The file that contains these is called fc3tuples<alias>.bin. For Alice this would be fc3tuplesalice.bin, for instance. Feralcore.java periodically writes the contents of the fc3tuplecontainer to fc3tuples<alias>.bin.

So, given a fresh installation of feralcore, how do we jumpstart the process of letting a new core "know" about other cores? Hold that thought. It is covered at the end of this section.

Feralcores send "fcsend" objects to one another. An fcsend object is either a program, ping, or ping response. The fcsend class is Fcsend.java. These objects get serialized and then deserialized. When a fcsend object is created on bob's node, the feralcore kernel places its own fc3tuple and up to two randomly selected fc3tuples into the fcsend object. When the object is sent to Alice's node, her kernel extracts these fc3tuples and adds them to her fc3tuplecontainer provided that certain conditions are met (e.g., an fc3tuple cannot have a timestamp that is too far in the future). So, when an fcsend program, ping, or ping response is sent, knowledge of f3tuples is spread throughout the feralcore network.

This design helps nodes discover other nodes. But, suppose a node goes down. Then since it will have fc3tuples lingering in the fc3tuplecontainers of other cores, it will be the equivalent of a "black-hole". Programs will be sent to it and never be seen or heard from again. To deal with this, feralcore has two strategies that try to eliminate fc3tuples from lingering on unnecessarily. These strategies are called from within the infinite loop in feralcore.mainloop() that runs the core.

Feralcore uses a pingwait strategy defined in Pingwait.java. pingwait removes fc3tuples from the tuplecontainer whose corresponding feralcore nodes do not respond quick enough to a feralcore ping. It pings and then waits for a response. It is careful to process ping requests while waiting for its ping response. pingwait does not rely on the validity of timestamps in received fc3tuples at all.

In addition, a pingnowait strategy is used. The Pingnowait.java class removes fc3tuples from the tuplecontainer that are deemed to be too old. It pings a remote feralcore node but does not wait for a response. tuplecontainer-obj.deletethreshold() is used to sweep away old fc3tuples from the tuplecontainer. pingnowait relies on the validity of timestamps in fc3tuples.

Despite the pingwait strategy, it is very important to run feralcore on a machine with an accurate real-time clock. Failure to do so will cause the feralcore node to do a poor job of maintaining the fc3tuplecontainer object.

Like programs that transmit themselves, feralcore ping packets tote up to 3 fctuples: the fc3tuple of the sending node and up to two other randomly selected f3tuples. This improves the ability of feralcore nodes to maintain awareness of other feralcore nodes.

There are a couple ways of jump-starting a new feralcore node so that it can join an existing feralcore network. Suppose Alice has a fresh installation and Bob is part of an existing feralcore network. Alice could ask Bob to send her a program. Her feralcore can learn up to 3 fc3tuples this way. Alternatively, Alice could send a program to Bob. Then his feralcore node will learn about Alice's and may end up sending programs her way.

4 Quick Start

In this quick-start you will setup and run two feralcores on your machine. True to crypto parlance, these will be run by "alice" and "bob". A number of files and cryptographic keys will be created. The files are all created in the same directory as the .java source files.

Sometimes when starting a feralcore program you may be told that entropy is being gathered. When this happens, type keys randomly or move and click the mouse randomly. This usually makes the entropy gathering go faster.

4.1 Quick Start for Generating Key Files

This section gives the commands for generating feralcore and fcdebug configuration files. It is intended for users that have done it before and want a concise listing of the process. For other folks, it is strongly recommended that you skip to Subsection 4.2 for a more detailed explanation.

The below steps are the files required for the feralcore node.

1. java Feralcorecfg
2. choose option (a) to generate the configuration files
3. enter the alias (e.g., **alice**), a password, port number, and Internet address
4. hit keyboard, move mouse *randomly*
5. files feralcorecfgpriv**alice**.bin, feralcorecfgpub**alice**.bin, and transto**alice**.bin are output

The below steps are the files required for fcdebug that talks to the node. The password you enter below should not be the same as the password used above in Feralcorecfg. The other values you enter must be the same as the ones you entered into the Feralcorecfg program.

1. java Debugcfg
2. choose option (a) to generate the configuration files
3. enter the same alias (e.g., **alice**), a different password, the same port number, and the same Internet address
4. files debugcfg**alice**.bin and debugcfg**alice**.bin are created and stored in two different places

That completes the creation of key files needed for feralcore and its fcdebug interface.

4.2 Create Key Files for Alice and Bob

In the feralcore source directory type "javac Feralcorecfg.java". Now type "java Feralcorecfg". This program generates the identity of a feralcore node: it's network address, port, private key, and so on. An *alias* is used to denote the feralcore node. In this case, the alias is "alice".

Choose option (a) to generate the feralcorecfgpriv<alias>.bin configuration file. You will be asked to enter information to create the file. You will also have to choose a password. You should make the password at least 16 characters. It will not accept short passwords. The following is the configuration that we used: "alice" as the alias, 33020 as the port, IPv4 as the type of address, 127.0.0.1 as the address. Key generation can take a long time. Bang on the keyboard/mouse to move it along.

Feel free to change the port if it conflicts with a port being used in your system. IPv6 support has not been fully implemented. You may want to use your actual IP address instead of the loop-back address 127.0.0.1. This will allow you to, e.g., have two nodes run in your local network on different machines.

When key generation completes you should have 3 new files. They are created in the same directory as the feralcorecfg program. The files are called "feralcorecfgprivalice.bin", "feralcorecfgpubalice.bin", and "transtoalice.bin".

The file feralcorecfgprivalice.bin contains the private key of Alice's feralcore node. This is what enables her feralcore to decipher programs sent to it. This is the only file among the 3 that has secret key information in it. This file must reside in the same directory as Feralcore.java. The private key it contains is protected by encryption and a password based key derivation function (PBKDF).

The file "feralcorecfgpubalice.bin" contains almost the same information, except that it contains the public key of Alice's node, not the corresponding private key. Giving this to others enables them to send encrypted messages to Alice's node.

Finally, a feralcore program was created. It is called "transtoalice.bin". Bob can load this program into his feralcore and run it. When he does, it will send itself to Alice's feralcore node. The program contains all the information needed to locate Alice's feralcore node on the Internet. In particular, it contains Alice's fc3tuple. This program is a convenient way for Bob to teach Alice's feralcore node about the presence of Bob's feralcore node on the Internet.

If you suspect you made a typo while creating "feralcorecfgprivalice.bin" you can check it. Run feralcorecfg and select the option that lets you display the contents of this file.

To complete the setup for Alice, one more key file must be created. Alice interacts with her feralcore node through the fcdebug program. fcdebug communicates with feralcore over UDP. This is a one-to-one interaction, so public key crypto is not needed. Consequently, this channel is encrypted using Blowfish-EDE in CBC mode with 3 128-bit keys. The key file for this channel will now be created.

Run "javac Debugcfg.java". Then run "java Debugcfg". Select option (a) to generate the file debugcfg<alias>.bin. Enter the alias "alice" and choose a long password randomly and enter it. Enter the same port number and address that you used to construct "feralcorecfgprivalice.bin". The file "debugcfgalice.bin" will be created. This file must be located in the directory containing Fcdebug.java and also the directory containing Feralcore.java. If this directory is one and the same then only one copy of this file is needed.

This completes the key generation for Alice. Now do the same for Bob, i.e., use the alias "bob". If you will be running Alice's feralcore on the same machine as Bob's feralcore, then you must use a different port for Bob. You may use 32020 as the port for Bob, for example. Do not forget which passwords correspond to which files.

4.3 Port Number Restrictions

There are three places where restrictions on the port numbers are imposed: your firewall, the Security Manager policy that is applied to the program (e.g., feralcore, fcdebug), and the program itself (e.g., feralcore, fcdebug). The policies that restrict port numbers are `feralcore.policy` and `fcdebug.policy`. Check those using a text editor or `policytool` to see the port restrictions.

Both the `Feralcore.java` and `Fcdebug.java` programs use port limitations defined in `Feralcoreconstants.java`. In particular the constants below define the minimum and maximum port numbers that feralcore and fcdebug can use.

```
Feralcoreconstants.MIN_PORTNUM
Feralcoreconstants.MAX_PORTNUM
```

If you run into issues with port use check all of these places.

4.4 File Access Restrictions

Feralcore and its related programs are designed to carefully utilize the Java Security Manager. The following file permissions are used: `read`, `write`, and `delete`. The `execute` permission is not utilized. The data files that are read, written, and deleted are organized so that file permissions can be applied in a fine-grained manner. The Feralcore.java program uses directories dataferalcorer, dataferalcorerw, dataferalcorew, and dataferalcorewd. The feralcore.policy file enforces the following access upon these, respectively: `read`, both `read` and `write`, `write`, and both `write` and `delete`. The Fcdebug.java program has its own set of directories and Rerandtrans.java has its own set of directories. Therefore, directories are first allocated in an application-specific fashion. This prevents applications from mucking with the files used by other applications. Then, directories are used that provide a specific combination of permissions. This enables permissions to be tailored to each file used by each application.

Scott Oaks book titled "Java Security", 2nd edition, on page 22 indicates that Java code can always read files within the directory *hiearchy* in which they are loaded. Therefore, the whole hierarchy ${user.dir} is readable by Java programs. For this reason all the feralcore data, fcdebug data, rerandtrans data, and policy directories are located at the same level in the file system as ${user.dir}. This prevents this "auto-read" permission from being granted to these directories.

4.5 Running the `imp`

Check out the assembly code for the imp in imp.asm. It is a program that copies itself to the adjacent memory region just above the program itself and then sends control to it. When it reaches the highest memory addresses it seemlessly copies and runs itself back at the lowest memory addresses. It will run forever until killed.

The assembled version of imp.asm is imp.bin. This is the raw program. It has position independent code. So, you can copy it anywhere in memory and it will run as long as the first instruction is on a word boundary. This holds even if the program straddles high memory to low memory.

Open two command line terminals side-by-side. Make sure both are in the feralcore `src` directory. In the left one type "javac Feralcore.java". In the right one type "javac Fcdebug.java". They should compile without error.

In the right terminal type the below command.

```
java -Djava.security.manager
       -Djava.security.policy=policies/fcdebug.policy
       Fcdebug alice <password>
```

The password must correspond to `debugcfgalice.bin`. A security manager is required. fcdebug will terminate with an exception if one is not installed. In an ideal world we would use "==" so that fcdebug.policy is the only policy file used. However, when this was done on Mac OS X Snow Leopard an exception was thrown. It appeared to be related to permissions associated with 2D graphics. So, for the fcdebug policy we use a single "=" above to supplement the default policy with fcdebug.policy.

You may have to bang on the keyboard/click and move the mouse until it gathers all the entropy it wants. When it is up and running, type "h1" and hit enter to view some commands. In the left terminal type in the below command:

```
java -Djava.security.manager
       -Djava.security.policy==policies/feralcore.policy
       Feralcore norelay none alice <password1> <password2>
```

Feralcore will not run without a security manager installed. Running it with the default security policy will cause it to throw an exception very quickly when it tries to write out a data file. So, always be sure to use the two -D flags noted above. The == tells the JVM to make feralcore.policy the only policy used. "norelay" tells the feralcore not to echo the programs it receives to a predefined port on the localhost. "none" tells the injector to produce rerandtrans programs that have no recipient. The third argument to feralcore is the alias of the feralcore you want to run. It must match the alias in "feralcorecfgpriv<alias>.bin" and "debugcfg<alias>.bin". The parameter <password1> is the password corresponding to "feralcorecfgpriv<alias>.bin". The parameter <password2> is the password corresponding to "debugcfg<alias>.bin". Wait a few moments until the PBKDF computations and entropy gathering complete.

The following are the commands that you will be asked to enter into the fcdebug window. They are stated here for reference. They are described below in detail.

1. session
2. loadprog imp.bin
3. sendprog `ffffffe0`
4. freerunoff
5. create `ffffffe0`
6. debug
7. Now type “n” and hit return several times
8. freerunon

Enter the command “session” into the fcdebug window followed by the enter key. All commands are executed by hitting enter. This command must be entered in order to send commands to a feralcore node. It helps protect the session from packet replay attacks (but is not perfect). Enter the command “loadprog imp.bin”. It should indicate that the loaded program has a length in bytes of 0000000020. This is the size of imp.bin in decimal. The program has been loaded into memory in fcdebug. Now enter the command “sendprog `ffffffe0`”. A message should indicate that the command completed successfully. The imp program has now been copied into the feralcore at address `0xffffffe0`. Now enter “freerunoff”. This command tells feralcore that we want to single-step through the program. Now enter “create `ffffffe0`”. This tells feralcore to create a new process with an initial value of of `ffffffe0` in the PC. This associates the new process with the imp program. fcdebug should respond with a process ID expressed in base64. To prevent users from having to type in lengthy process IDs, the create command copies it into a variable called “buffer”. Now enter the command “debug”. This is shorthand for “debug <processID>”. It takes the processID from the value in the buffer.

At this point fcdebug should show you the disassembly of imp.bin. The first instruction is LEA. The asterisk in front of LEA tells you that this is the next instruction to be executed. Issue the “n” command many, many times. Watch the imp as it copies and runs itself seamlessly from high memory to low memory. When you are satisfied with the correctness of imp.asm, issue the “freerunon” command. Watch as the feralcore goes crazy executing the imp.

When you terminate fcdebug it does not affect feralcore. However, if feralcore is not left in free-running mode then it will not process programs properly. If you terminate fcdebug and bring it back up, you will need to start a new session using the “session” command.

When you are done observing the imp, you should shut down fcdebug and feralcore. You terminate fcdebug using the Swing GUI. In Ubuntu there is an “x” button in the upper left hand corner of the window that closes the window. In Ubuntu you can kill feralcore by entering ctrl-c.

4.6 Running `transtoalice`

The purpose of this exercise is to show how Bob can teach his feralcore node about the existence of Alice’s node and how to contact it. As a result, Bob’s fc3tuplecontainer will contain an fc3tuple of Alice. In the process Alice’s node will learn an fc3tuple of Bob’s node.

In the event you have experimented previously, you need to delete two files for a clean run. If present, delete `fc3tuplesalice.bin` and `fc3tuplesbob.bin`.

These files preserve the contents of the fc3tuplecontainer when the feralcore is not running. If your fc3tuples file ever becomes corrupt it can simply be deleted. However, you will need to jump-start your core again to teach it about the presence of at least one other active feralcore node. The file will grow in size automatically over time.

The first step is for Bob to get a hold of the program file `transtoalice.bin`. Alice could post this file on her website, for instance. Or, she could e-mail it to Bob.

It is assumed that you have already compiled feralcore and fcdebug. Create 4 terminal windows. Make sure they are all in the `src` directory. In them run Alice's feralcore, Alice's fcdebug, Bob's feralcore, and Bob's fcdebug. Be sure not to get Alice's fcdebug mixed up with Bob's fcdebug. Use "norelay" and "none" as in subsection 4.5. In Bob's fcdebug window execute the following commands in succession: "session", "loadprog transtoalice.bin", "sendprog `101010fe`", "create `101010fe`". There is nothing special about the address `0x101010fe` except that it is even. These commands do not debug the program. They copy it into the core and run it.

Now sit back and watch as the transtoalice program causes the feralcore nodes to discover one another. This process will take a few mintues. Keep tabs on the size of the files `fc3tuplesalice.bin` and `fc3tuplesbob.bin`. They should go from a size of 0 to a size of several hundred bytes. When both have a non-zero length, then the nodes have memorized one another. The feralcore program periodically writes the contents of its fc3tuplecontainer object to disk.

The program kills itself when it arrives at Alice's node. But by the time it does, it will have accomplished its goal. It carried Bob's fc3tuple with it to Alice's node. Alice's feralcore kernel extracts this fc3tuple automatically and remembers it. After some time, Alice's node will send a feralcore ping Bob's node to see if it is still up. This ping carries with it Alice's fc3tuple. This is when Bob's node learns of Alice's node.

You can compile and run `Fc3tuplecontainer.java` to inspect the contents of "fc3tuples<alias>.bin" . You have to pass it the alias at the command line.

4.7 Running `slowhopper`

This exercise assumes that you just completed the exercise in subsection 4.6. Therefore, the file `fc3tuplesalice.bin` contains only one fc3tuple, one representing Bob's feralcore. Also, `fc3tuplesbob.bin` contains only one fc3tuple, one representing Alice's feralcore. This exercise will be clear unless this is the case.

Run a node for Bob and connect an fcdebug session to it. Run a node for Alice. Immediately enter the commands that follow after both nodes have gathered all the entropy they want. Don't dilly-dally, because if you do, the injectors in the nodes will start injecting programs and you won't notice your slowhopper process by manual inspection.

Enter the following commands in succession in the fcdebug window that is connected to Bob's core: "session", "loadprog slowhopper.bin", "sendprog `101010fe`", "create `101010fe`". Now watch as the slowhopper chills out in a busy-waiting fashion, sends itself to Alice's node, chills out, sends itself to Bob's node, chills-out, and so on. You will know when the injector programs kick-in.

To see why it behaves as it does, read the MC68000 code in `slowhopper.asm`. It lives up to it's name.

Slowhopper would be a great tool for checking the stability of your network connection between Alice and Bob. But, the injector obfuscates the slowhopper's activity. If you want, disable the injector at the source code level and run the slowhopper for a day or so. Convince yourself that your connectivity is stable.

5 Assembling Programs

We used the vasm assembler system by Volker Barthelmann to produce feralcore programs. Dr. Volker has released an open-source C compiler and assembler that produces MC68000 programs. The assembler compiles very easily in Ubuntu. It can generate the raw binary of an MC68000 program and save it to a file which is very convenient. Dr. Volker's vasm website is listed below.

```
http://www.compilers.de/vasm.html
```

At the time of this writing, the download is at the URL below. The source file is `vasm.tar.gz`.

```
http://sun.hasenbraten.de/vasm/release/vasm.tar.gz
```

To build the assembler executable `vasmm68k_mot` in, e.g., Ubuntu use the Makefile that comes with vasm. Execute the following in the directory containing the Makefile:

```
make vasmm68k_mot CPU=m68k SYNTAX=mot
```

The assembly programs that are passed to `vasmm68k_mot` must use Motorola syntax. This is due to the use of the argument "mot" during the creation of `vasmm68k_mot`.

The command below assembles `imp.asm` to produce the imp program in the raw binary file `a.out`. This is how the imp program was created. Simply rename `a.out` to `imp.bin`. Note that the imp consists of position-independent instructions. So, you can load the file `imp.bin` into fcdebug, send it to any even-numbered address in the core, and it will run just fine.

```
./vasmm68k_mot -Fbin imp.asm
```

Sometimes it is useful to compile C code to see what the assembly source looks like. The open-source tarball `vbcc.tar.gz` contains the vbcc compiler. You can compile `helloworld.c` to produce `helloworld.asm` as shown below.

```
vbccm68k.exe "helloworld.c" -o="helloworld.asm"
                -Ic:/vbcc/targets/m68k-ataritos/include
```

You might need to comment out things like "opt 0" and "opt NQLPSMR" in `helloworld.asm` to get `vasmm68k_mot` to assemble `helloworld.asm`.

6 YYRerandTrans

YYRerandTrans is an end-to-end message posting protocol in which Alice can post a message to a bulletin board. It is an example application that is built on top of feralcore. The mechanics of this protocol is also a good foundation for simply sending private messages from Alice to Bob. This was implemented in the C++ feralcore but is not currently implemented in the Java feralcore.

In a nutshell, YYRerandTrans operates like this. Bob runs a bulletin board implemented using a website. The website has a CGI program that accepts messages to be posted. Alice gets a hold of Bob's public key. She encrypts a message with it. It becomes the immediate operand of the API call `APICallReRandTrans`. The program consists of this single instruction. She submits it to the feralcore network. When run, this instruction probabilistically re-encrypts its immediate operand and then sends the resulting modified instruction to a randomly selected feralcore node.

Bob's feralcore node is executed using the `relay` argument. So, his node echos all received programs to a port on his localhost. Bob runs a program called `Rerandtrans.java` that reads that port and deciphers the `APICallReRandTrans` programs that it receives using Bob's private key. When it receives a rerandtrans program bound for someone else, decryption will simply fail. When it receives Alice's message, it will decrypt it.

`rerandtrans` collects the plaintexts it receives in a pool. At certain time intervals it chooses a random subset of the pool and sends this subset to the bulletin board CGI program using HTTP POST.

The CGI program has a limit to the number of messages it will post to the website. When too many have been received the oldest are deleted to make room for the new.

The `APICallReRandTrans` encryption used in the immediate operand has these properties:

1. It includes a cryptographic integrity check. This mitigates the risk of the plaintext being modified in an undetected fashion.
2. It is re-randomizable without first deciphering it and without revealing the recipient's public key. Probabilistic re-encryption is a well-known notion in cryptography.
3. It contains a timestamp. If `Rerandtrans.java` receives a message in a relayed program that is too far in the past or future then it is dropped. This helps protect against replay attacks.

6.1 Using YYRerandTrans

This subsection does not explain how to setup a bulletin board. It assumes that a bulletin board is up and running and that you want to post a message to it.

We recommend that the public key for a bulletin board be posted next to the hyperlink to the bulletin board. We also recommend that the public key file be signed and that the signature be posted along with the public key file.

So for example, suppose that you want to post a message on Bob's bulletin board. The first step is to go to his bulletin board and make sure it is active. If you go to the webpage you will see a creation time. Make sure it is recent. Then, download the public key file needed to post to the bulletin board.

An example bulletin board is given below.

`http : //www.feralcore.com/messages.html`

Files containing a rerandtrans public key are named "pubkey<alias>.key". Here `alias` should conventionally be the name of the person running the feralcore node corresponding to the yyrerandtrans bulletin board.

Now enter the command "javac Rerandtrans.java". This will compile the rerandtrans program. Then enter the command "java Rerandtrans". This runs the rerandtrans program. Now enter command (c) to encrypt a message and produce a rerandtrans program file. In yyrerandtrans the message you send is a feralcore program containing the encryption of your message.

You will be asked to enter the alias in the public key file name. For example, if Bob is running the yyrerandtrans bulletin board and his public key file is called pubkeybob.key you would enter "bob". Enter the plaintext message that you want to post. To see the list of invalid characters, see `isvalidchar()` in `Rerandtrans.java`. If you enter a valid message, the file "rerand<alias>.bin" will be created. This file is created in the same directory as the rerandtrans program. Using our example, the file "rerandbob.bin" would be created.

Now, submit this program several times to your feralcore node. If one of these programs makes it to Bob's node fast enough it will be added to the pool of messages there. Eventually, the message will be posted on the web.

6.2 The Crypto of YYRerandTrans

YYRerandTrans utilizes what is known as probabilistic re-encryption. For work related to probabilistic re-encryption see [22]. Re-encryption will now be reviewed.

Let p, q, and g be as defined in Section 2. Let $\mathbb{Z}_q$ denote the set of non-negative integers less than q. All operations are modulo p. Choose x randomly from $\mathbb{Z}_q$. $y = g^x$ is an ElGamal public key and x is the corresponding private key.

Encryption: The plaintext is m, a quadratic residue modulo p. The method for encrypting octet streams is addressed below. Choose k_1, k_2 randomly from $\mathbb{Z}_q$. Compute $c = (a_1, b_1, a_2, b_2) = (g^{k_1}, y^{k_1}, g^{k_2}, y^{k_2} * m)$. Output the ciphertext c.

Decryption: Ignore a_1 and b_1 in c. Compute $m = \frac{b_2}{a_2^x}$. Output m.

The cool thing about the 4-tuple c is that it is simultaneously an encryption of m and an encryption of y. This hides the identity of the intended recipient of c while allowing c to be re-randomized by anyone. The process of re-encrypting c is called *re-encryption.*

Re-encryption: Choose t_1, t_2 randomly from $\mathbb{Z}_q$. Compute $c_2 = (a_3, b_3, a_4, b_4)$ as follows.

$$\begin{aligned} a_3 = a_1^{t_1} &= g^{k_1 * t_1} \\ b_3 = b_1^{t_1} &= y^{k_1 * t_1} \\ a_4 = a_2 * a_1^{t_2} &= g^{k_2 + k_1 * t_2} \\ b_4 = b_2 * b_1^{t_2} &= y^{k_2 + k_1 * t_2} * m \end{aligned}$$

Observe that the pair (a_1, b_1) has been rerandomized and the pair (a_2, b_2) has been rerandomized. The pair (a_4, b_4) is a valid encryption of m. Output the ciphertext c_2.

We stated that the plaintext is a quadratic residue. So, how then could one encrypt an octet stream? Feralcore handles this problem by shrinking the cardinality of the effective set of plaintexts, employing a nonce during encryption, and using a pseudorandom permutation and its inverse. See the code in `YYcrypto.java` for the implementation.

A modified form of probabilistic re-encryption is used in yyrerandtrans to encrypt a message and form the immediate operand of `APICallReRandTrans`. The details of the actual implementation are not be covered here.

7 API Reference

The operation word `0x0008` was added to the MC68000 instruction set to implement feralcore API calls. The format of a feralcore API call is the word `0x0008` followed by the API vector. For some instruction there may be immediate data following the vector. The vector uniquely identifies the API call that is being made.

For example, in the syntax of the vasm assembler, the below is a call to APICallWriteRegB().

```
blk.w 1,$0008   ; make APICall
blk.w 1,$0001   ; APICallWriteRegB()
```

A vector that does not correspond to an implemented API call implies that the API call is an illegal instruction. When encountered, the feralcore kernel will terminate the calling feralcore process. This is a good way to get your program to terminate deliberately. Below is an illegal instruction since `0x100A` is undefined.

```
blk.w 1,$0008   ; make APICall
blk.w 1,$100A   ; invalid vector so terminate program
```

Register B is a large data register created to support the feralcore protocol. This register is written to and read using feralcore API calls. The main purpose of this register is to store an fc3tuple since it is so large and is used frequently. The length in bytes of this register is fc3tuple.serializedlength(). This is specified in the Java source file `Fc3tuple.java`.

7.1 Vector `0x0000`: APICallIsImplemented

The input is the lower 16 bits of D0. This value represents the API vector in question. Upon completion, D0 = 1 if the input vector is an implemented feralcore API call and D0 = 0 otherwise.

7.2 Vector `0x0001`: APICallWriteRegB

The input is register A0 that points to the data in the core to write to register B. The fc3tuple.serializedlength() bytes in the core pointed to by A0 are written to register B. There is no output.

7.3 Vector `0x0002`: APICallReadRegB

The input is register D0 that contains an index into register B. D0 must be between 0 and fc3tuple.serializedlength()-1 inclusive. The output is placed in D0. The uppermost 3 bytes are 0x00. The least significant byte of D0 is the byte that was read. If the index is invalid then `0xFFFFFFFF` is returned in D0.

By allowing only one byte to be read we prevent this call from being used to blast large sections of core memory. Had we permitted the whole register to be dumped into core memory, this API call could be used to very effectively terminate processes running in the core. So, the byte level read is used to ensure fairness in feralcore.

7.4 Vector `0x0003`: APICallSizeFC3TupleContainer

There is no input for this API call. The output is the number of fc3tuples in the fc3tuplecontainer object that is placed in D0. This is the size of fc3tuplecontainer object.

7.5 Vector `0x0004`: APICallGetFC3Tuple

The input is an index into the fc3tuplecontainer object in D0. The index must be between 0 and the size of the fc3tuplecontainer object - 1, inclusive. The output is a serialized fc3tuple in register B on success. `0xFFFFFFFF` is returned in D0 if the call fails (index is invalid). `0x00000000` is returned in D0 if the call succeeds.

7.6 Vector `0x0005`: APICallGetRandomFC3Tuple

There is no input for this API call. The output is placed in register B and D0. When it succeeds, a randomly selected fc3tuple is placed in register B. It is in serialized form. `0xFFFFFFFF` is returned in D0 if the call fails. `0x00000000` is returned in D0 if the call succeeds. The call fails if there are no fc3tuples in the container.

7.7 Vector `0x0006`: APICallRandom

There is no input for this API call. The output is a random 32-bit value in D0.

7.8 Vector `0x0007`: APICallCreate

The input to this call is register A0 that contains the address of a program. There is no output. A process is created with the PC set to the input value in A0. All registers except the PC are set to zero initially in the new process. Also, the CPU flags are set to false in the new process.

7.9 Vector `0x0008`: APICallTrans

A process invokes this API call to transmit a program to a feralcore node. A0 contains the address of the program in the core that will be sent to the destination core. D0 is the length of the program being sent. The program must be at least 2 bytes and at most Feralcoreconstants.MAX_PROGRAM_SIZE bytes in length. A1 is the starting address in the destination core that the program will be copied to. Register B contains the fc3tuple defining the destination feralcore.

Upon completion D0 is set to `0xFFFFFFFF` if the transmission fails. If the transmission succeeds then there is no return value since the calling process gets killed.

7.10 Vector `0x0009`: APICallGetFCCounter

Feralcore maintains an object of type fcounter defined in `Fccounter.java`. This is a counter that is related to the uptime but is randomized. Upon construction this counter generates and stores a random 24-bit value called `randoffset`. Class fccounter is immutable so `randoffset` does not change after the constructor terminates. The output counter value, when queried, is `randoffset` plus the current uptime measured in seconds. The uptime is calculated to be ManagementFactory.getRuntimeMXBean().getUptime()/1000. When this API call is made, the current value of the fccounter object is returned in D0. This call is useful for measuring elapsed time.

7.11 Vector `0x000A`: APICallReRandTrans

This API call is an instruction with an immediate operand. The instruction atomically re-encrypts its immediate operand and then sends itself to a randomly selected feralcore node. Therefore, the program that is sent is a single instruction, this API call. The destination feralcore node is selected uniformly at random from the fc3tuplecontainer object. The re-encryption algorithm is covered in Section 6.

There is no input and no output for this API call. The immediate operand is yycrypto.RERAND_CTEXT_LEN bytes long. It follows the opword vector for this call. So, this instruction is the opword for a feralcore API call followed by the opword vector followed by the rerand ciphertext.

7.12 Vector `0x000B`: APICallFeralcoreVersion

This API call returns the version of the feralcore source in register D0. For example, if the source code tar ball is `yyjfcore2213.tar.gz` then 2213 will be returned in D0. Of course, there is nothing to stop a feralcore node from misrepresenting in this API call the version of feralcore that is running.

8 Locked-Down Machines

This is one suggestion for running a locked down Fedora machine. It was tested with Fedora 16.

Make sure selinux is enabled. Read the file /etc/sysconfig/selinux. Alternatively, issue the `getenforce` command. selinux was not enabled in a linode instance we tested, a Fedora 15 instance. A note online indicated that selinux is a compile-time option that is disabled by default in linode kernels to achieve compatability with non-selinux systems. With some effort it may be possible to setup a linode instance running selinux.

See what ports are listening ports. Get rid of most of them. Run the below as root or with sudo:

```
netstat --tcp --listening --programs

netstat --udp --listening --programs
```

If the avahi-daemon is running then disable it by doing the following as root (or with sudo):

```
systemctl disable avahi-daemon.service
```

If UDP port ipp is being listened to then remove cups. Do the following as root (or with sudo):

```
yum remove cups
```

If sendmail is listening in on a tcp port then do the following as root (or with sudo):

```
yum remove sendmail
```

The above 3 removals diminished the listening ports on FC16 to leave only ntp listening ports remaining.

9 Conclusion

This concludes our description of the feralcore protocol. This is an evolving document that we expect will grow and improve over time. We hope you enjoy experimenting with feralcore.

References

1. C. Adami, C. T. Brown. Evolutionary Learning in the 2D Artificial Life Systems Avida. In *Proc. Artificial Life IV*, MIT Press, pages 377–381, 1994.
2. Aleph-Null. Computer Recreations. In *Software: Practice and Experience*, vol. 2, pages 93–96, 1972.
3. L. von Ahn, N. J. Hopper. Public-Key Steganography. In *Advances in Cryptology—Eurocrypt '04*, pp. 323–341, 2004.
4. J. Bloch Effective Java. Second Edition, Addison-Wesley, 2008.
5. A. Charrel. Tierra network version. ATR Technical Report TR-H-145, 1995.
6. A. K. Dewdney. In the game called Core War hostile programs engage in a battle of bits. In *Scientific American*, May 1984.

7. T. El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *IEEE Trans. Info. Theory*, 31(4):469–472, 1985.
8. N. J. Hopper, J. Langford, L. von Ahn. Provably Secure Steganography. In *Advances in Cryptology—Crypto '02*, pp. 77–92, 2002.
9. R. J. Jenkins Jr. ISAAC. In *Fast Software Encryption*, pages 41–49, 1996.
10. B. Möller. A Public-Key Encryption Scheme with Pseudo-Random Ciphertexts. In *Proc. of ESORICS '04*, pp. 335–351, 2004.
11. T. S. Ray. A proposal to create a network-wide biodiversity reserve for digital organisms. ATR Technical Report TR-H-133, 1995.
12. T. Ray, C. Xu, A. Charrel, T. Kimezawa, T. Yoshikawa, M. Chaland, T. Uffner. Tierra Documentation. Tierra.doc 12-27-2000 documentation for the Tierra Simulator, Tierra Simulator V6.02.
13. M. Santha, U. V. Vazirani. Generating quasi-random sequences from slightly-random sources. In *Proceedings of the 25th IEEE Symposium on Foundations of Computer Science*, pages 434–440, 1984.
14. B. Schneier. Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In *Fast Software Encryption*, pages 191–204, 1993.
15. R. C. Seacord. Secure Coding in C and C++. Addison-Wesley, 2005.
16. J. F. Shoch, J. A. Hupp. The "Worm" Programs: Early Experience with a Distributed Computation. In *Communications of the ACM*, vol. 25, issue 3, pages 172–180, 1982.
17. J. M. Smith, S. M. Nettles. Active Networking: One View of the Past, Present and Future. In *IEEE Transactions On Systems, Man and Cybernetics, Part C: Applications and Reviews*, vol. 34, no. 1, pages 4–18, 2004.
18. D. L. Tennenhouse, D. J. Wetherall. Towards an Active Network Architecture. In *Computer Communication Review*, 1996.
19. Y. Tsiounis, M. Yung. On the security of ElGamal-based encryption. In *Proc. Public Key Cryptography—PKC '98*, pp. 117-134, 1998.
20. A. Young, M. Yung. The Dark Side of Black-Box Cryptography, or: Should We Trust Capstone? In *Advances in Cryptology—Crypto '96*, pp. 89–103, 1996.
21. A. Young, M. Yung. Malicious Cryptography: Exposing Cryptovirology. John Wiley & Sons, Inc., Feb., 2004.
22. M. Jakobsson. A Practical Mix. In *Advances in Cryptology—Eurocrypt '98*, pages 448–461, 1998.