TinyMAMA: Mobile-agent Authenticated Migration for Agilla *

Final Report

Robert Grant
University of Texas at Austin
bgrant@mail.utexas.edu

Agoston Petz
University of Texas at Austin
agoston@mail.utexas.edu

ABSTRACT

Wireless sensor networks receive a great deal of attention in current research. They have the potential to be widely deployed because they are appropriate for a large number of applications, both civilian and military. However, like any wireless network, they suffer from a wide range of security problems, and these problems are exacerbated by their computational constraints. Many standard security solutions are impossible or impractical due to their resource requirements, and existing sensor network security protocols could be improved by controlling them from a level that has more contextual knowledge. We created TinyMAMA, a lightweight authentication protocol for mobile agents on top of TinyOS and Agilla that exposes low level security primitives to agents, which are in a better position to intelligently control security trade-offs.

General Terms

mobile agent, authentication, sensor network

1. INTRODUCTION

Wireless sensor networks (WSNs) receive a great deal of treatment in modern research. Appropriately so, since they are useful in a large number of situations where remote sensing devices are required, but for which existing technology cannot provide a solution. WSNs usually consist of small, cheap, networked devices with at least one on-board sensor that can gather environmental information. The devices are able to communicate their sensor readings to other devices using an available wireless network, and together the devices form a distributed data collection tool. WSNs have a multitude of applications including military intelligence, home and manufacturing automation, and physical security [2]. Typical sensor devices are easy to deploy and can be combined to provide a heterogeneous, highly flexible, and capable infrastructure. However, some of the advantages of wireless sensor networks are also impediments to the programmers who write software for them. Sensor devices are usually severely constrained in terms of processing power, on-board memory, battery life, networking range, and bandwidth [8]. Implementing security protocols on sensors thus becomes very difficult, as care must be taken to minimize resource usage.

In this paper, we limit our focus to wireless sensor networks that accommodate mobile agents (Agilla [3], Maté [13], and Magnet [8] are good examples of such systems). A mobile agent can be loosely defined as code that can change its execution location based on its environment or some other dynamic stimulus [4]. Mobile agents provide an unprecedented level of flexibility in the design and deployment of wireless sensor networks. They allow programmers to insert new software into an existing system thus allowing them to dynamically modify it or to introduce entirely new functionality both easily and cheaply. Furthermore, mobile agents are able to perform many functions that traditional distributed systems would be unable to accomplish given the severe bandwidth constraints of WSNs [4].

Our contribution, TinyMAMA, adds a simple, lightweight authentication protocol to Agilla by which a platform can authenticate a migrating mobile agent (hereafter simply referred to as an "agent") before allowing it to run. This is important to any application that requires security since, if a platform is not careful, a malicious agent could possibly subvert it or cause resource starvation. Furthermore, malicious agents can inject bad data into a network or prevent legitimate agents from carrying out their tasks. There are additional attacks which we will not cover—please see [10] for a more thorough treatment.

We use the Agilla mobile agent framework (or middleware) to implement our project. It was designed specifically for wireless sensor networks, it provides a simple API for developing agents, and it is one of the only mobile agent middleware systems that is currently maintained and publicly available. Agilla runs on TinyOS, a popular and widely-deployed operating system which runs on a wide range of sensor devices [3]. While there are several WSN-specific authentication protocols available, some of which run on TinyOS, none of them are particularly flexible. None of them provide a means by which agent-to-platform authentication can take place independent of other security associations and interactions. They treat authentication as a binary property, either

*This space intentionally left blank

on or off, whereas our solution puts the choice in the hands of the mobile agent. We think this is important since the mobile agent is privy to contextual information and thus in a better position to make decisions. Authentication is expensive since it generally requires more information to be sent with the packets, and a single additional bit of transmitted data uses the same amount of battery power as executing 800-1000 instructions [12]. There is a trade-off between security and battery power [16], and it makes sense to allow the mobile agent to decide this trade-off dynamically.

Our implementation provides an Agilla agent with two additional instructions, `asmove` and `awmove`, which are authenticated variants of the standard `smove` and `wmove` instructions. Both instructions make use of pre-shared symmetric keys to provide the basis of trust. This simple and rather naïve mechanism could be easily enhanced to make use of more flexible root-of-trust mechanisms based on secure hardware.

The remainder of this paper is organized along the following lines: in Section 2 we provide an overview of related research, in Section 3 we cover our design, in Section 4 we give a thorough overview of our implementation, and we discuss possible future work in this area in Section 5.

2. RELATED WORK

We did not find any prior work that deals specifically with mobile agent authentication in sensor networks, which is where our work is focused. However, we did find several projects and proposals that are related. These can be split into two categories: work on mobile agent security and work on sensor network security. Most of these proposals use symmetric key cryptography, but we also include a small section on projects that are trying to bring asymmetric key cryptography to sensor networks.

2.1 Mobile Agent Security

Many papers that discuss mobile agent security do not consider resource constrained devices. Some of the tactics they propose, while valid for general purpose PCs, are complex and difficult to use in sensor networks, while others could be applied to sensors. A paper by W.A. Jansen of NIST [9] provides a good overview of many of the techniques that we review in this section.

Of the techniques designed to protect the agent platform, *sandboxing*, *safe code interpretation*, *state appraisal*, and *proof carrying code* are all probably too memory hungry or too computationally complex to use under tight resource constraints. The technique of *code signing* is very similar to what we are doing. Jansen describes code signing in the context of using asymmetric key cryptography to digitally sign a mobile agent so that the author of the agent can be verified. Since asymmetric key cryptography is computationally intensive, we use a Message Authentication Code (MAC) based on a shared symmetric key. Though a particular author cannot be authenticated using a MAC (since at least two people possess the shared key), it can be verified that the author is someone who knows the secret key, and is therefore "trusted". Additionally, since a particular author cannot be uniquely identified without using asymmetric key cryptography, the idea of a *path history*, or a verifiable chain of host platforms, is also infeasible.

Many of the schemes that protect agents also rely on asymmetric key cryptography, or otherwise assume resources not available to mobile agents in sensor networks. In particular, itinerary-based schemes such as *mutual itinerary recording* and *itinerary recording with replication and voting* require the ability to uniquely verify platforms and the ability to communicate with other agents through authenticated channels. The second is difficult in ad-hoc networks in general; the first is difficult without public and private keys. The other schemes presented, such as *execution tracing*, attempt to validate the functionality of a mobile agent. If this is even possible in a processor constrained sensor node, it would leave few resources for performing actual work.

Some systems that fall under this category are the D'Agents system out of Dartmouth [6] which requires Tcl or Scheme interpreters on every node, and a proposal by G. Vigna that uses execution tracing [18].

2.2 Sensor Network Security

There is other work in sensor network security that does not specifically consider mobile agents, and we leverage some of this work for our system.

2.2.1 SPINS

The SPINS security protocol suite encompasses two protocols, SNEP and μTESLA. SNEP is a protocol for encrypted, authenticated, point-to-point communication, and μTESLA is a protocol for secure authenticated broadcast.

SPINS is based on the assumption that asymmetric key cryptography is infeasible in sensor networks, so it uses symmetric key algorithms based on a shared key to do authentication and encryption. The basic encryption algorithm used in this system is RC5, and a Cipher Block Chaining MAC (CBC-MAC) is used for authentication. One original contribution in their system is a consideration for "freshness" in their authentication mechanism.

The SPINS algorithms depend on high powered base stations coordinating communication among sensor nodes. This is reasonable in some scenarios, but it does not generalize to ad-hoc sensor networks. It also does not scale well—especially the proposed methods for authenticated broadcast by a sensor.

The authors provide performance measurements based on an implementation on top of TinyOS, but the authors of TinySec [12] claim that SPINs was never fully implemented. In any case we were unable to find a publicly accessible version of SPINs. This was enough to remove it from consideration as a building block for our system.

2.2.2 LiSP

There is an interesting proposal by a group from the University of Michigan called LiSP: A Lightweight Security Protocol for Wireless Sensor Networks [15]. They also consider asymmetric key cryptography to be infeasible on sensor networks. Their idea is to split the network into groups or clusters, each with at least one node, the Group Head (GH), that is capable of more complex processing. They employ a hierarchy of keys, employ different ciphers at different times,

and use an Intrusion Detection System to detect when nodes have been compromised. They also include the ability to re-key sensors when a compromise has been detected. Unfortunately, there does not seem to be an implementation available.

2.2.3 TinySec

TinySec, a project out of UC Berkeley, is the work most relevant to our own effort. It is a fully implemented link layer security architecture that is now shipped with TinyOS. The system is also surprisingly efficient—the authors claim that it adds "less than 10% energy, latency, and bandwidth overhead" [12].

TinySec is designed to be transparent and easy to use. As shown in the TinySec manual [11], there are three modes—TinySec-AE (authentication and encryption), TinySec-Auth (only authentication), or both disabled. When enabled, these cryptographic primitives are applied frame by frame, and provide either access control, message integrity, confidentiality, or a combination of these. The authors claim that TinySec is cipher independent (they have implemented RC5 and Skipjack), and they assume that symmetric keys are pre-shared, or are distributed through some external key distribution system.

The creators of TinySec intend for it to be used as a basis for higher level security protocols. We have done so, and leverage much of the work they did, especially regarding their efficient implementation of Message Authentication Codes (MACs). Please see section 4.2.1.

Since TinySec is a link layer architecture, it does not apply its security mechanisms very intelligently. When any of the security modes are enabled, security is applied to every frame transmitted and received. No matter how efficient the provided security schemes are, the overhead of these mechanisms could be reduced by applying them more discriminantly at a higher layer.

Additionally, the authors of TinySec intentionally ignore replay attacks, which are attacks that involve recording protected data and retransmitting it with malicious intent. Though the contents of this data may not be known to an attacker, this could still be a very disruptive attack in a sensor network, especially on availability. It may also be possible to handle this at a higher layer.

2.3 Asymmetric Key Cryptography

Some projects claim the ability to do public key cryptography on sensor nodes, such as TinyPK [19] which uses low exponent RSA, and a group from Harvard [14] that has implemented Elliptic Curve Cryptography for TinyOS. While these projects show that such things are possible, they are still very slow, or require some of the work to be done by more powerful third parties.

3. DESIGN

In this section we present our design of the authenticated migration mechanism. We cover the actual authentication protocol in section 3.1, and the means by which authenticated agents are delivered and verified in sections 3.2 and 3.3.

3.1 Authentication Mechanism

In order for an agent to authenticate itself to a platform, it must carry proof that it is migrating from a trusted source. This proof is easily added to an agent by signing the agent with a key that the target platform trusts. The platform can simply verify that the agent was signed by the key, and decide whether or not to run the agent based on the result of this verification. Signature schemes are often implemented using public/private key pairs, however as we discussed earlier, asymmetric key cryptography is not feasible on our small, resource constrained nodes. Symmetric key cryptography can also be used to create signatures by using Message Authentication Codes (MACs), and this is the method we chose for our implementation. Since the key is the same for both the sender and the verifier, a pre-shared key distribution system is perhaps the simplest and easiest way to establish a network of trust. There are a number of well known flaws in this mechanism. For example, if a single node is compromised, the secret key of the entire network is compromised. There are a number of ways the key distribution mechanism could be improved, as we discuss in Section 5.4.

Since TinySec provides an efficient implementation of symmetric block ciphers, we decided to use a Cipher Block Chaining MAC (CBC-MAC) for authentication. It requires a symmetric block cipher, which is executed on the message block by block. Each output block is then `XOR`'d with the next input block to ensure that the resulting MAC is dependent on every bit the message [17]. Please see Figure 1 for a graphical representation of this.

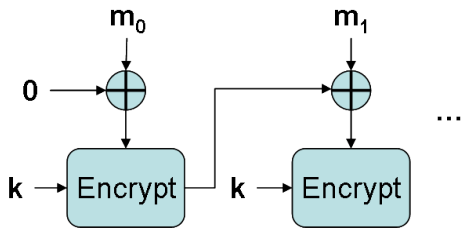

Figure 1: CBC-MAC

Our underlying authentication mechanism is based on the work done by Karlof, Sastry, and Wagner for TinySec. They determined that a length of four bytes for the MAC is appropriate for sensor networks since it provides a good trade-off between overhead and security [12]. In fact our code leverages the cryptographic primitives exported by TinySec's implementation of the Skipjack CBC-MAC. The entire agent, including code and context, is used as input to the MAC. In keeping with our design philosophy of putting the trade-off in the hands of the user, we designed our code to be flexible, and thus changing the length of the desired MAC is a simple modification. Changing from Skipjack to an RC5-based CBC-MAC is also a very simple modification.

3.2 Delivery

Once the MAC is calculated over the entire agent, the resulting four-byte value is `push`'d on to the agent's stack before the agent is sent to the target platform. This method allowed us to leverage the already efficient "agent-packing"

code available in Agilla. Since an authenticated agent appears just like an un-authenticated agent, we did not have to create any special frame formats or message types to accommodate our new instructions.

3.3 Verification

To verify that a received agent is authenticated before it is allowed to run, we first reassemble the agent. Once the agent is intact, the MAC can be `pop`'d off the agent's stack, it can be recalculated locally, and the result can be compared to the `pop`'d value. If they match, the target platform is assured that the agent was not modified in transit, and that it originates from a trusted source. If the MAC does not match, the agent is thrown out. It is slightly inefficient to completely reassemble the agent before checking whether or not it is authenticated. If the agent is malicious, then a lot of time was wasted for no good reason. It would be better if the platform learned whether not to trust the source before reassembling the agent, and even better if it could learn this information before accepting all of the packets involved in the transmission of an agent (thus saving precious radio time and battery life). However, we feel that our solution is a good trade-off between overhead, and implementation complexity. Furthermore, if the malicious party's main goal is to waste resources such as battery life, there are much simpler ways to accomplish this.

4. IMPLEMENTATION

In this Section we present our implementation of the authenticated migration instructions. We provide a brief overview of the sensor platform we used in Section 4.1, and a detailed description of how we added our authentication mechanism to Agilla in Section 4.2. We also describe how we tested our solution is Section 4.3 and discuss the problems we faced implementing the authenticated move and how we fixed them in section 4.4.

4.1 Hardware

We implemented TinyMAMA on Crossbow Mica2 Sensor Motes [1]. The Mica2 motes feature a 16MHz ATmega128L 8-bit microcontroller and 128KB of program memory. The particular motes we used have 315 MHz radios. The motes were running TinyOS version 1.1.15, and we implemented our code on Agilla version 3.0.2, the latest packaged release of Agilla. To set the radio frequency to match our motes we modified the default Agilla compiler flags by adding the following line to `Makelocal`:

```
PFLAGS += -DCC1K_DEF_FREQ=315178000
```

4.2 Modifying Agilla

Since Agilla is implemented in the nesC programming language [5], our authentication mechanism is also implemented in nesC. The bulk of our code lives in two Agilla modules located in the Agilla code directory: `opcodes/OPmoveM.nc` and `components/AgentMgrM.nc`. The following two sections explain how we modified Agilla to send authenticated agents, and how we modified it to receive authenticated agents. Our modifications are available as a patch to Agilla version 3.0.2 in, which is included in Appendix B.

4.2.1 Sending Authenticated Agents

`OPmoveM` contains the code which calculates the MAC of the agent and `push`'s the result on the agent's stack before handing the agent off to the `AgentMgr` for packaging and transmission. `OPmoveM` is the first module which is called when an agent executes any kind of move instruction, thus it is the ideal location for our code. In order to avoid duplicating functionality, we use the CBC-MAC implementation exported by TinySec, specifically TinySec's `MAC` interface. It handles variable size inputs, and can produce variable size outputs so long as the requested MAC length is a multiple of the cipher's block size (one byte in this case).

We calculate the MAC of the entire agent which consists of two parts: the fixed-size data structure which holds the agent's context (registers, stack, heap) and a linked list of code fragments which contains all of the agent's code. Figure 2 shows the contents of the agent's context data structure—note that the MAC is not calculated over the entire contiguous contents of the data structure, but instead calculated in two parts. The first MAC begins at the top of the data structure and contains everything up to and including the last *in-use* byte of the agent's stack. The second MAC begins after the last byte of the agent's stack and covers the rest of the context. This is done for two reasons. First, it would be wasteful to calculate the MAC over the entire contents of the stack since the only data that the agent will ever use lies between the top of the stack and the current location of the stack pointer. Authenticating the rest of the stack is pointless since anything between the stack pointer and the end of the stack is garbage. Second, there is a possibility that the agent packing code which lies below `OPmoveM` in the Agilla stack might now, or in the future, optimize the network transmission by throwing out the part of the stack that the agent is not currently using, and we want to make sure our code is compatible in this case.

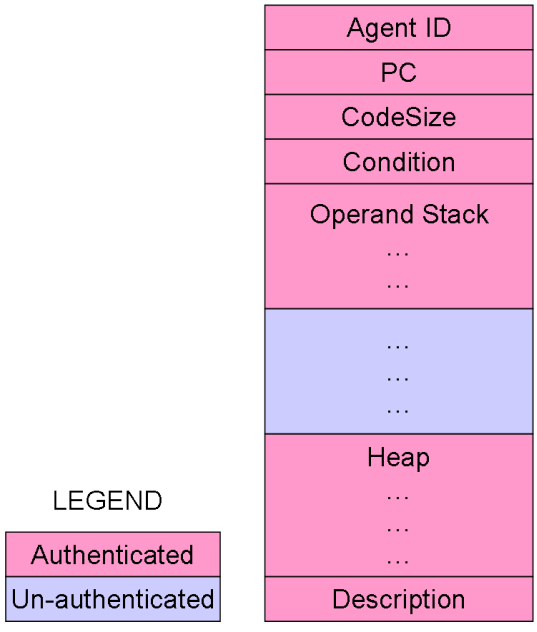

Figure 2: Agent Context Authentication

TinySec's MAC calculating function expects a contiguous block of memory for its input, therefore we calculate two independent MACs for the agent context. Also, since code blocks (which are 22 bytes long) are not necessarily contiguous, we calculate an independent MAC for each block. All of the resulting intermediate MACs are `XOR`'d together to create the final MAC of the agent. This technique is fast, efficient, and maintains the cryptographic integrity of the authentication code.

After the MAC is calculated, the resulting four-byte value is `push`'d onto the agent's stack like any other variable. This method is an easy and efficient way to package the MAC with the agent, but there are two potential problems. First, if the agent is making full use of its stack already, the migration will fail since the `OPmoveM` module will be unable to `push` the MAC. Second, if the agent is executing an authenticated *weak* move (`awmove`) the stack will not be sent since it is cleared anyway before the agent is restarted. We do not consider the first problem to be of any real significance since it is not unreasonable to force the agent to clear four bytes of its stack before migrating. We address the second problem by forcing the agent to execute a strong migration even when it calls `awmove`. In this case we modify the context before calculating the MAC in the following ways: the program counter of the agent is set to 0 to ensure that the agent will restart from the beginning when run on the target node, and we set the stack pointer to 0 to clear the agent's stack. There is more overhead associated with our authenticated weak migration than a regular weak migration since more context must be sent. However, the underlying Agilla code should take care of optimizing the transmission by sending only the part of the stack that is in use (which in this case is only the part that holds the MAC).

4.2.2 Receiving Authenticated Agents

We handle receiving agents in much the same way as sending them. The first module in the Agilla stack which has access to the fully reassembled agent is the `AgentMgrM`, thus it is an ideal place for our verification code. First, the MAC is `pop`'d off the stack, then it is independently recalculated in the same way as before. The results are compared, and the agent is only run if the calculated MAC matches the received MAC. It would be more efficient to determine if the agent is authenticated *before* reassembling it; please see Section 3.3 for a discussion of this.

4.3 Testing

In order to test our implementation we used TOSSIM to simulate motes and injected agents that would hop between them. We extensively instrumented the code with debug statements, and it is our hope that most of these have been removed from the released patch. We did additional testing on Mica2 motes. We did not have time for performance testing, so we were unable to compare the performance of our authenticated move instruction to any existing alternatives (such as TinySec). However, we feel that our overhead is quite low given that agents span multiple packets and TinySec will calculate a four-byte MAC over each packet. Our implementation only sends one four-byte MAC (packed in a five-byte data structure) for an entire agent, which spans multiple packets. Ways in which performance could be evaluated are discussed in Section 5.1. Furthermore, we did not change TinyOS's packet format so our solution is more interoperable than TinySec. The agent code we used to test the solution on Mica2 motes is available in Appendices A.1 and A.2.

4.4 Problems

During our implementation we discovered a bug in TinySec's implementation of CBC-MAC. Specifically the bug is in the version of TinySec which ships with TinyOS 1.1.15, and it is in the function `MAC.incrementalMAC` in the file `tos/lib/TinySec/CBCMAC.nc`. It causes TinySec to crash if a buffer larger than 255 bytes is used as input to the MAC function. We fixed this bug and include a patch in Appendix C.

There are a number of known problems with our implementation. First, it is vulnerable to replay attacks since our mechanism does not provide a method for ensuring live-ness. Second, our mechanism does not protect agents from malicious platforms; it only protects platforms from malicious agents, and prevents malicious third parties from injecting agents into an existing network. However, if an agent migrates to a platform which lacks the shared key, it will not be able to rejoin the trusted network. It would be nice to prevent an agent from migrating to a un-trusted platform in the first place. This could have a devastating affect on an application if the application programmers have not accounted for this possibility. We discuss possible solutions to these problems in Future Work, please see Section 5

5. FUTURE WORK

TinyMAMA is already both useful and functional in its current state, but there is room for improvements. The next step could be a performance analysis of our instructions to compare them with both unauthenticated migration and with migration authenticated at the link layer with TinySec. We also have a number of ideas for extending TinyMAMA, which we present in this section. These include adding authenticated clone instructions, adding the ability to encrypt a mobile agent's data payload, key distribution using asymmetric key cryptography, and incorporating a three-way handshake to protect against replay attacks and to allow mobile agents to authenticate platforms. We also discuss the advantages of using a hardware root-of-trust mechanism.

5.1 Performance Analysis

Though we have extensively simulated our instructions in TOSSIM, performance tests on real motes would be useful to help quantify TinyMAMA's trade-offs between security, power usage, performance, and ease-of-use. One informative performance test would be an analysis of how long it takes an agent to hop across a series of nodes. We could compare the speed difference between our strong and weak authenticated move functions, and between TinyMAMA authenticated agents and agents authenticated at the link layer with TinySec. The speed of the original, unauthenticated move instructions could be used as a normalization factor.

5.2 Authenticated Agent Cloning

In addition to agent migration instructions (`smove` and `wmove`), Agilla supports agent duplication instructions (`sclone` and

`wclone`). The clone instructions share a large part of their code with the move instructions in the Agilla code base, so adding authenticated versions of these instructions should be a simple extension to TinyMAMA's current functionality.

5.3 Encrypted Agent Payload

Another useful extension (or complementary project) to TinyMAMA would be adding the ability for a mobile agent to encrypt its data payload. This would prevent a passive attacker from reading sensitive data from the agent while it is in transit. Such a project would continue the philosophy on which TinyMAMA is based, that mobile agents should have the ability to control security functionality (like authentication and encryption) from a high level so it can be used in an appropriate and targeted manner. This project could also leverage security primitives from TinySec, but would probably be a more invasive addition than TinyMAMA.

5.4 Asymmetric Key Extensions

The current implementation of TinyMAMA relies exclusively on pre-shared symmetric keys and does not consider the problem of key distribution. However, as mentioned in Section 2.3, there have been a number of projects that have developed asymmetric key cryptography specifically for key distribution in sensor networks [14, 19]. TinyMAMA could easily leverage the shared symmetric keys provided by these systems. However, interesting future could explore whether specially designed public key cryptographic algorithms could be used in agent authentication. This could provide a much finer-grained authentication capability than currently available in TinyMAMA. Though current evidence suggests that such a scheme would be extremely slow, it is worth exploring.

5.5 Three-Way Handshake

To prevent malicious third-parties from simply recording authenticated agent migrations and retransmitting them bit by bit (a replay attack), one could extend TinyMAMA's authentication instructions to implement a basic three-way handshake. This method can also be used to allow a mobile agent to authenticate a platform, which is not currently supported by TinyMAMA. There is more overhead associated with a three-way handshake than with a simple timestamp-based signature, but three-way handshakes can prevent replay attacks without relying on synchronized clocks or synchronized counters. This is an important characteristic since some sensors will be unable to keep synchronized clocks without modification, and keeping counters synchronized in a poorly connected network requires a fair amount of resynchronization overhead.

Figure 3 shows the messages involved in such an exchange. All messages in the three-way handshake are signed using a MAC which could use the same CBC-MAC algorithm and the same shared key that TinyMAMA already uses for mobile agent authentication.

A brief overview of the three-way handshake messages follows.

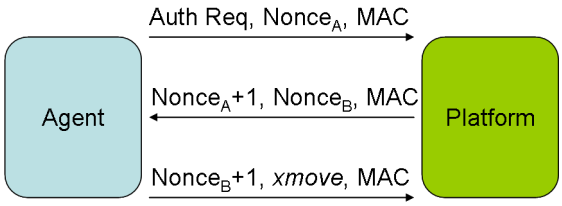

Figure 3: Three-Way Handshake Protocol

1. The agent begins the protocol by sending an Authentication Request to the platform along with a nonce, both signed by a CBC-MAC which uses the shared symmetric key.

2. The platform checks the MAC to ensure that the agent knows the secret key. It then sends back the agent's nonce+1, and its own nonce, both signed by the CBC-MAC.

3. The agent checks the MAC to ensure that the platform knows the secret. It then checks to see if the platform sent back the correct nonce+1 thereby ensuring that the platform is live and thwarting replay attacks. If everything checks out, the agent begins an authenticated migration, includes the platform's nonce+1, and signs the message with the CBC-MAC.

When the platform receives the last message, it can verify that the agent is live, and if the MAC is correct it accepts the migration and sends back an ACK as appropriate.

As always, the security of the protocol relies on the mobile agent's and the platform's ability to keep the shared secret private, which leads to our next section.

5.6 Hardware Root of Trust

Sensors, especially those deployed in WSNs, are vulnerable to physical attacks as well as software-based attacks. The nodes are often physically accessible, hard to keep track of, and generally not tamper-resistant. This makes it difficult to establish a root-of-trust since you cannot be sure whether a node is compromised or not. Tamper-proof hardware is one way to solve this problem, but it is often an expensive and impractical solution. However, a hardware root-of-trust architecture such as the one developed by the Trusted Computing Group (TCG) for PCs [7] could be implemented on sensors as well. The root-of-trust is established using a small, relatively inexpensive, tamper-proof computing device that has very limited capabilities. This small device can be used to "bootstrap" trust by authenticating the first piece of code (often an operating system loader or BIOS of some kind) before allowing it to run. This loader can then authenticate the next piece of code, and so on. This process is sometimes called *inductive trust* and it could be applied to sensor networks. If a system similar to the TCG architecture was designed around the constraints of sensor nodes, it could be used to establish semantic tamper-proofness at a much lower cost.

6. CONCLUSION

Although there are still open issues with our implementation, it is usable in its current state. It can also be easily integrated with existing Agilla agents, simply by changing their `wmove` and `smove` instructions to our authenticated variants. Additionally, TinyMAMA makes it easier to deploy performance-conscious, security-enabled applications by leveraging the Agilla middleware. Our work can be easily extended to expose additional security primitives to mobile agents through the Agilla ISA.

Ultimately, we feel that TinyMAMA is a useful addition to the Agilla sensor network middleware, and a step in the right direction for sensor network security in general. It provides a lightweight, authentication mechanism that is easily controlled by mobile agents, which are privy to contextual information that allows them to intelligently decide the tradeoff between security and performance. It is therefore more flexible than any previous WSN authentication scheme, and helps bridge the gap between mobile agent security, and sensor network security.

`THE END`

7. REFERENCES

[1] Mica2 Datasheet. http://xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

[2] C. Chong, S. Kumar, and B. Hamilton. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.

[3] C. Fok, G. Roman, and C. Lu. Mobile agent middleware for sensor networks: an application case study. *Proceedings of the 4th international symposium on Information processing in sensor networks*, 2005.

[4] A. Fuggetta, G. Picco, G. Vigna, and D. e Inf. Understanding code mobility. *Software Engineering, IEEE Transactions on*, 24(5):342–361, 1998.

[5] D. Gay, P. Levis, D. Culler, and E. Brewer. nesC 1.1 Language Reference Manual. *TinyOS documentation site, available in http://today. cs. berkeley. edu/tos/tinyos-1. x/doc/nesc/ref. pdf, May*, 2003.

[6] R. Gray, D. Kotz, G. Cybenko, and D. DAgents. Security in a multiple-language, mobile-agent system in: Mobile Agents and Security. *Lecture Notes in Computer Science*, 1419, 1998.

[7] T. C. Group. TCG Specification Architecture Overview, Revision 1.2. https://www.trustedcomputinggroup.org/groups/TCG_1_0_Architecture_Overview.pdf, 2004.

[8] S. Hadim and N. Mohamed. Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3), 2006.

[9] W. Jansen. Countermeasures for mobile agent security. *Computer Communications*, 23(17):1667–1676, 2000.

[10] W. Jansen and T. Karygiannis. NIST Special Publication 800-19–Mobile Agent Security. *Gaithersburg, MD: National Institute of Standards and Technology, August*, 1999.

[11] C. Karlof, N. Sastry, and D. Wagner. TinySec: User Manual. *Manuscript, September*, 12:9, 2003.

[12] C. Karlof, N. Sastry, and D. Wagner. TinySec: a link layer security architecture for wireless sensor networks. *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, 2004.

[13] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review*, 36(5):85–95, 2002.

[14] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 71–80, 2004.

[15] T. Park and K. Shin. LiSP: A lightweight security protocol for wireless sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):634–660, 2004.

[16] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. *Security Protocols, 7th International Workshop*, 1999.

[17] W. Stallings. *Cryptography and Network Security: Principles and Practices*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2003.

[18] G. Vigna. Cryptographic Traces for Mobile Agents. *Mobile Agents and Security*, 1419:137–153, 1998.

[19] R. Watro, D. Kong, S. Cuti, C. Gardiner, C. Lynn, and P. Kruus. TinyPK: securing sensor networks with public key technology. *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*, pages 59–64, 2004.

APPENDIX

A. MOBILE AGENTS

A.1 Agilla Agent 1 - Continuous Hop between 0 and 1

```
 1: BEGIN pushc 10
 2:       sleep
 3:       pushc 28
 4:       putled       // toggle the yellow LED
 5:       pushc 5
 6:       sleep        // sleep for 1/4 second
 7:       pushc 28
 8:       putled       // toggle the yellow LED
 9:       addr
10:       pushc 0
11:       cneq
12:       pushc MIG0
13:       jumpc
14: MIG1  pushc 1
15:       asmove
16:       pushc BEGIN
17:       jumpc
18:       pushc ERROR
19:       jumpc
20: MIG0  pushc 0
21:       asmove
22:       pushc BEGIN
23:       jumpc
24: ERROR pushc 25
25:       putled       // toggle yellow LED when
26:       pushc 20     // migration fails
27:       sleep
28:       pushc 25
29:       putled
30:       halt
```

A.2 Agilla Agent 2 - Tries Un-Authenticated Hop

```
 1: BEGIN  pushc 10
 2:        sleep
 3:        pushc 28
 4:        putled       // toggle the yellow LED
 5:        pushc 5
 6:        sleep        // sleep for 1/4 second
 7:        pushc 28
 8:        putled       // toggle the yellow LED
 9:        pushc 1
10:        asmove
11:        pushc 10
12:        sleep
13:        pushc 28
14:        putled       // toggle the yellow LED
15:        pushc 5
16:        sleep        // sleep for 1/4 second
17:        pushc 28
18:        putled       // toggle the yellow LED
19:        pushc 0
20:        asmove
21:        pushc 10
22:        sleep
23:        pushc 28
24:        putled       // toggle the yellow LED
25:        pushc 5
26:        sleep        // sleep for 1/4 second
27:        pushc 28
28:        putled       // toggle the yellow LED
29:        pushc 1
30:        smove
31: END    halt
```

B. PATCH FOR AGILLA 3.0.2

```
diff -Naur Agilla-orig/Agilla.nc Agilla-ours/Agilla.nc
--- Agilla-orig/Agilla.nc 2006-12-14 12:28:16.138315200 -0600
+++ Agilla-ours/Agilla.nc 2006-12-07 00:25:49.000000000 -0600
@@ -173,6 +173,7 @@
   AgillaEngineC.BasicISA[IOPsetvars] -> OPgetsetvars;
   AgillaEngineC.BasicISA[IOPdec] -> OPdec;

+
   #if !ENABLE_CLUSTERING
     AgillaEngineC.BasicISA[IOPclearvar] -> OPclearvar;
     AgillaEngineC.BasicISA[IOPcisnbr] -> OPcisnbr;
@@ -564,6 +565,11 @@
   AgillaEngineC.ExtendedISA1[IOPesetvar+61] -> OPegetsetvar6;
   AgillaEngineC.ExtendedISA1[IOPesetvar+62] -> OPegetsetvar6;
   AgillaEngineC.ExtendedISA1[IOPesetvar+63] -> OPegetsetvar6;
+
+  // Bob and Tony's instructions
+  AgillaEngineC.ExtendedISA2[IOPasmove] -> OPmove;
+  AgillaEngineC.ExtendedISA2[IOPawmove] -> OPmove;
+
   #endif
 }

diff -Naur Agilla-orig/AgillaOpcodes.h Agilla-ours/AgillaOpcodes.h
--- Agilla-orig/AgillaOpcodes.h 2006-12-14 12:28:16.148329600 -0600
+++ Agilla-ours/AgillaOpcodes.h 2006-12-07 00:25:10.000000000 -0600
@@ -153,6 +153,7 @@
   IOPsense     = 0x1f,
   IOPdec       = 0x20,

+
   // Two and three operand-instructions
   IOPdist      = 0x21,
   IOPswap      = 0x22,
@@ -253,4 +254,11 @@

 } ExtendedISA1;

+
+typedef enum {
+  // Bob and Tony's instructions
+  IOPasmove    = 0x00,
+  IOPawmove    = 0x01,
+} ExtendedISA2;
+
 #endif
diff -Naur Agilla-orig/Makefile Agilla-ours/Makefile
--- Agilla-orig/Makefile 2006-12-14 12:28:16.719150400 -0600
+++ Agilla-ours/Makefile 2006-12-04 00:47:52.000000000 -0600
@@ -14,7 +14,8 @@
        -I types \
        -I interfaces \
        -I ../SpaceLocalizer \
-       -I ../LEDBlinker
+       -I ../LEDBlinker \
+       -I $(TOSROOT)/tos/lib/TinySec

 #MSG_SIZE=27
```

```
diff -Naur Agilla-orig/Agilla.nc Agilla-ours/Agilla.nc

diff -Naur Agilla-orig/components/AgentMgrC.nc Agilla-ours/components/AgentMgrC.nc
--- Agilla-orig/components/AgentMgrC.nc 2006-12-14 12:28:16.158344000 -0600
+++ Agilla-ours/components/AgentMgrC.nc 2006-12-05 23:54:13.000000000 -0600
@@ -72,6 +72,8 @@
  * be found here: http://www.gnu.org/copyleft/lesser.html
  */
 includes Agilla;
+includes crypto;
+includes CryptoPrimitives;

 /**
  * Manages agent contexts.
@@ -87,6 +89,7 @@
 }
 implementation {
   components AgentMgrM, AgillaEngineC, CodeMgrC, NeighborListProxy;
+  components CBCMAC, SkipJackM as Cipher;

   #if OMIT_AGENT_SENDER
     components AgentSenderDummy as AgentSender;
@@ -144,4 +147,8 @@
   #if ENABLE_EXP_LOGGING
     AgentMgrM.ExpLoggerI -> ExpLoggerC;
   #endif
+  AgentMgrM.MAC -> CBCMAC.MAC;
+
+  CBCMAC.BlockCipher -> Cipher.BlockCipher;
+  CBCMAC.BlockCipherInfo -> Cipher.BlockCipherInfo;
 }
diff -Naur Agilla-orig/components/AgentMgrM.nc Agilla-ours/components/AgentMgrM.nc
--- Agilla-orig/components/AgentMgrM.nc 2006-12-14 12:36:36.798228800 -0600
+++ Agilla-ours/components/AgentMgrM.nc 2006-12-14 18:43:28.439414400 -0600
@@ -36,6 +36,8 @@
  * be found here: http://www.gnu.org/copyleft/lesser.html
  */

+// #define INCLUDE_DEFAULT_AGENT 1
+
 includes Agilla;
 includes AgillaOpcodes;  // for debugging purposes
 includes LEDBlinker;
@@ -73,6 +75,7 @@
     #if ENABLE_EXP_LOGGING
       interface ExpLoggerI;
     #endif
+    interface MAC;
   }
 }
 implementation {
@@ -93,6 +96,8 @@
     call AgentMgrI.run(context);
   }

+
+
   /**
    * Generate a new agent ID.
    */
```

```
@@ -114,7 +119,7 @@
       if (TOS_LOCAL_ADDRESS == 0)
       {
         #include "default_agent.ma"
-        //runNewAgent(&agents[0]);
+        runNewAgent(&agents[0]);
         call AgentSenderI.send(&agents[0], agents[0].id, IOPwmove, 1, 1);
        }
     #endif
@@ -143,6 +148,29 @@

   /*event result_t SimTimer.fired() {
     #if INCLUDE_DEFAULT_AGENT
+        agents[0].id.id = getNewID();
+        agents[0].pc = 0;
+        agents[0].state = AGILLA_STATE_READY;
+        call CodeMgrI.allocateBlocks(&agents[0], 15);
+        dbg(DBG_USR1, "||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
+        dbg(DBG_USR1, "Running our agent prepare to be bedazzled\n");
+        cMsg.msgNum = 0;
+        cMsg.code[0] = IOPpushc | 25;
+        cMsg.code[1] = IOPputled;
+        cMsg.code[2] = IOPpushc | 2;
+        cMsg.code[3] = IOPsleep;
+        cMsg.code[4] = IOPpushc | 25;
+        cMsg.code[5] = IOPputled;
+        cMsg.code[6] = IOPaddr;
+        cMsg.code[7] = IOPpushc | 0;
+        cMsg.code[8] = IOPcneq;
+        cMsg.code[9] = IOPrjumpc | 5;
+        cMsg.code[10] = IOPpushc | 1;
+        cMsg.code[11] = IOPwmove;
+        cMsg.code[12] = IOPpushc | 28;
+        cMsg.code[13] = IOPputled;
+        cMsg.code[14] = IOPhalt;
+        call CodeMgrI.setBlock(&agents[0], &cMsg);

     #endif
     return SUCCESS;
@@ -344,6 +372,94 @@
   {
     if (dest == TOS_LOCAL_ADDRESS) {

+
+     AgillaVariable macvar,tmp;
+     uint8_t key[16] = {0xAC, 0x68, 0xBA, 0x41, 0x95, 0xBE, 0x3D, 0xEE, 0x8B, 0x9D,
       0x69, 0x2C, 0x96, 0x19, 0xE5, 0x63};
+     uint8_t keySize = 16, codeSize =0;
+     MACContext maccontext;
+     uint8_t i=0,k=0,j=0;
+     uint8_t codeCtr = 0;
+     uint8_t mac[4] = {0,0,0,0};
+     uint8_t mac1[4] = {0,0,0,0};
+     uint8_t mac2[4] = {0,0,0,0};
+     uint8_t mac3[4] = {0,0,0,0};
+     uint8_t firstHashSize, secondHashSize;
+     uint8_t macv[4] = {0,0,0,0};
+     uint16_t* mac16 = 0;
+     uint8_t data[4] = {1,2,3,4};
+     static uint8_t migrationCtr = 0;
+     uint8_t instr;
+     uint8_t codeBlock[AGILLA_CODE_BLOCK_SIZE];
+     uint8_t done = 0;
+     uint8_t tmpmac[4] = {0,0,0,0};
```

```
+
+
+     if(migrationCtr > 0) {
+     dbg(DBG_UART, "authenticated move %d\n",migrationCtr);
+      /* Bob and Tony's security code
+       * check agents MAC before running it
+       * if MAC fails, do not allow agent to run */
+      call OpStackI.popOperand(context, &macvar);
+      if (macvar.vtype != AGILLA_TYPE_READING) // this is a hack, we know
+        return FAIL;
+
+     call MAC.init(&maccontext, keySize, key);
+
+     firstHashSize = (context->opStack.sp)-1 + 8;
+     secondHashSize = 102;
+     call MAC.MAC(&maccontext, (uint8_t*)&(context->id),firstHashSize, mac1, 4);
+     call MAC.MAC(&maccontext, (uint8_t*)&(context->heap),secondHashSize, mac2, 4);
+
+/* *********  Calculate Code MAC ********** */
+
+  codeSize = context->codeSize;
+  dbg(DBG_USR1,"\tCCMAC: codeSize = %d bytes\n",codeSize);
+  while (!done) {
+    for (k=0; k<AGILLA_CODE_BLOCK_SIZE; k++) {
+      if (codeCtr < codeSize) {
+        instr = call CodeMgrI.getInstruction(context, codeCtr);
+        codeCtr++;
+      } else {
+        done = 1;
+        instr = 0;
+      }
+      codeBlock[k] = instr;
+    }
+    // code block written, calculate the mac.
+    call MAC.MAC(&maccontext, codeBlock, AGILLA_CODE_BLOCK_SIZE, tmpmac, 4);
+    for(j=0; j<4; j++) {
+      mac3[j] = mac3[j] ^ tmpmac[j];
+    }
+  }
+
+     for(k=0; k<4; k++) {
+        mac[k] = mac1[k] ^ mac2[k] ^ mac3[k];
+     }
+     dbg(DBG_USR1, "B&T(decrypt): MAC1 result = 0x%X%X%X%X\n",mac1[0],mac1[1],
+       mac1[2],mac1[3]);
+     dbg(DBG_USR1, "B&T(decrypt): MAC2 result = 0x%X%X%X%X\n",mac2[0],mac2[1],
+       mac2[2],mac2[3]);
+     dbg(DBG_USR1, "B&T(decrypt): MAC3 result = 0x%X%X%X%X\n",mac3[0],mac3[1],
+       mac3[2],mac3[3]);
+     dbg(DBG_USR1, "B&T(decrypt): MAC executed result = 0x%X%X%X%X\n",mac[0],mac[1],
+       mac[2],mac[3]);
+
+     mac16 = macv;
+     *mac16 = macvar.reading.type;
+     mac16 = &macv[2];
+     *mac16 = macvar.reading.reading;
+
+     /* check MAC */
+     for (k=0; k<4; k++) {
+        if (macv[k] != mac[k]) {
+          /* notify failure by flashing LED */
+         call LEDBlinkerI.blink((uint8_t)GREEN | YELLOW, (uint8_t)1, ARRIVAL_LED_TIME);
+          return FAIL;
+        }
+     }
+     }
```

```
+
+
       // send the agent migration trace to the base station
       #if ENABLE_EXP_LOGGING
         AgillaLocation loc;
@@ -351,9 +467,7 @@
         call ExpLoggerI.sendTraceQid(context->id.id, TOS_LOCAL_ADDRESS, AGENT_MOVED,
           TOS_LOCAL_ADDRESS, SUCCESS, loc);
       #endif

-      /*#ifdef PACKET_SIM_H_INCLUDED
-        uint8_t instr;
-        uint16_t i = 0;
+      //#ifdef PACKET_SIM_H_INCLUDED
         dbg(DBG_USR1, "Agent ID: %i\n", context->id.id);
         dbg(DBG_USR1, "Agent PC: %i\n", context->pc);
         dbg(DBG_USR1, "Agent CodeSize: %i\n", context->codeSize);
@@ -365,8 +479,11 @@
           if (i == context->codeSize)
             break;
         }
-      #endif*/
+      //#endif
+      dbg(DBG_UART, "authenticated move %d\n",migrationCtr);
+      migrationCtr++;
       runNewAgent(context);
+
     } else {
       uint16_t oneHopDest = dest;
       result_t forward = SUCCESS;
@@ -413,6 +530,8 @@
    * @return SUCCESS If the agent is scheduled to run.
    */
   command result_t AgentMgrI.run(AgillaAgentContext* context) {
+
+
     if (context->state != AGILLA_STATE_HALT) {
       context->state = AGILLA_STATE_RUN;
       return call AgentExecutorI.run(context);
diff -Naur Agilla-orig/components/ContextDiscovery/NeighborListM.nc Agilla-ours/components/
  ContextDiscovery/NeighborListM.nc
--- Agilla-orig/components/ContextDiscovery/NeighborListM.nc 2006-12-14 12:28:16.418718400 -0600
+++ Agilla-ours/components/ContextDiscovery/NeighborListM.nc 2006-11-20 00:52:24.000000000 -0600
@@ -442,7 +442,7 @@
       nbrs[indx].timeStamp = now;
       nbrs[indx].chId = bmsg->chId;
       nbrs[indx].energy = bmsg->energy;
-      nbrs[indx].linkQuality = m->lqi;
+      //nbrs[indx].linkQuality = m->lqi;

       #if ENABLE_CLUSTERING

diff -Naur Agilla-orig/components/NetworkInterface/NetworkInterfaceC.nc Agilla-ours/components/
  NetworkInterface/NetworkInterfaceC.nc
--- Agilla-orig/components/NetworkInterface/NetworkInterfaceC.nc 2006-12-14 12:28:16.508848000 -0600
+++ Agilla-ours/components/NetworkInterface/NetworkInterfaceC.nc 2006-11-20 00:51:59.000000000 -0600
@@ -52,7 +52,8 @@
 }
 implementation {
   components GenericComm as Comm, NetworkInterfaceM as NIM;
-  components MessageBufferM, CC2420RadioC;
+  //components MessageBufferM, CC2420RadioC;
+  components MessageBufferM;
   //components GenericCommPromiscuous as CommP, MultiHopRouter;

   #ifdef TOSH_HARDWARE_MICA2
```

```
@@ -77,7 +78,7 @@
   NIM.SendMsg -> Comm.SendMsg;
   NIM.ReceiveMsg -> Comm.ReceiveMsg;
   NIM.MessageBufferI -> MessageBufferM;
-  NIM.CC2420Control -> CC2420RadioC.CC2420Control;
+  //NIM.CC2420Control -> CC2420RadioC.CC2420Control;

   #ifdef TOSH_HARDWARE_MICA2
     NIM.MacControl -> CC1000RadioC;
diff -Naur Agilla-orig/components/NetworkInterface/NetworkInterfaceM.nc Agilla-ours/components/
  NetworkInterface/NetworkInterfaceM.nc
--- Agilla-orig/components/NetworkInterface/NetworkInterfaceM.nc 2006-12-14 12:28:16.508848000 -0600
+++ Agilla-ours/components/NetworkInterface/NetworkInterfaceM.nc 2006-11-20 00:51:23.000000000 -0600
@@ -72,7 +72,7 @@
     interface ReceiveMsg[uint8_t id];
     interface MessageBufferI;

-    interface CC2420Control;
+    //interface CC2420Control;
     #ifdef TOSH_HARDWARE_MICA2
       interface MacControl;     // enable MAC-level ACKs
       //interface CC1000Control;  // used to reduce the radio range
@@ -141,7 +141,7 @@
     }

     // set the transmit power
-    call CC2420Control.SetRFPower(AGILLA_RF_POWER);
+    // call CC2420Control.SetRFPower(AGILLA_RF_POWER);
     return SUCCESS;
   }

diff -Naur Agilla-orig/components/default_agent.ma Agilla-ours/components/default_agent.ma
--- Agilla-orig/components/default_agent.ma 1969-12-31 18:00:00.000000000 -0600
+++ Agilla-ours/components/default_agent.ma 2006-12-14 18:42:00.983659200 -0600
@@ -0,0 +1,44 @@
+        agents[0].id.id = getNewID();
+        agents[0].pc = 0;
+        agents[0].state = AGILLA_STATE_READY;
+        call CodeMgrI.allocateBlocks(&agents[0], 32);
+        dbg(DBG_USR1, "||||||||||||||||||||||||||||||||||||||||||||||||||||||\
+n");
+        dbg(DBG_USR1, "Running our_demo_agent.ma\n");
+        cMsg.msgNum = 0;
+        cMsg.code[0] = IOPpushc | 10;
+        cMsg.code[1] = IOPsleep;
+        cMsg.code[2] = IOPpushc | 28;
+        cMsg.code[3] = IOPputled;
+        cMsg.code[4] = IOPpushc | 5;
+        cMsg.code[5] = IOPsleep;
+        cMsg.code[6] = IOPpushc | 28;
+        cMsg.code[7] = IOPputled;
+        cMsg.code[8] = IOPaddr;
+        cMsg.code[9] = IOPpushc | 0;
+        cMsg.code[10] = IOPcneq;
+        cMsg.code[11] = IOPpushc | 20;
+        cMsg.code[12] = IOPjumpc;
+        cMsg.code[13] = IOPpushc | 1;
+        cMsg.code[14] = IOPextend2;
+        cMsg.code[15] = IOPawmove;
+        cMsg.code[16] = IOPpushc | 0;
+        cMsg.code[17] = IOPjumpc;
+        cMsg.code[18] = IOPpushc | 25;
+        cMsg.code[19] = IOPjumpc;
+        cMsg.code[20] = IOPpushc | 0;
+        cMsg.code[21] = IOPextend2;
```

```
+        call CodeMgrI.setBlock(&agents[0], &cMsg);
+
+        cMsg.msgNum = 1;
+        cMsg.code[0] = IOPawmove;
+        cMsg.code[1] = IOPpushc | 0;
+        cMsg.code[2] = IOPjumpc;
+        cMsg.code[3] = IOPpushc | 25;
+        cMsg.code[4] = IOPputled;
+        cMsg.code[5] = IOPpushc | 20;
+        cMsg.code[6] = IOPsleep;
+        cMsg.code[7] = IOPpushc | 25;
+        cMsg.code[8] = IOPputled;
+        cMsg.code[9] = IOPhalt;
+        call CodeMgrI.setBlock(&agents[0], &cMsg);
diff -Naur Agilla-orig/opcodes/OPmove.nc Agilla-ours/opcodes/OPmove.nc
--- Agilla-orig/opcodes/OPmove.nc 2006-12-14 12:28:16.919438400 -0600
+++ Agilla-ours/opcodes/OPmove.nc 2006-12-06 23:51:16.000000000 -0600
@@ -38,6 +38,8 @@


 includes Agilla;
+includes crypto;
+includes CryptoPrimitives;

 configuration OPmove {
   provides interface BytecodeI;
@@ -45,6 +47,8 @@
 implementation {
   components OPmoveM, OpStackC, AgentMgrM, ErrorMgrProxy;
   components LocationMgrC, NeighborListProxy, AddressMgrC;
+  components CBCMAC, SkipJackM as Cipher;
+  components CodeMgrC;

   BytecodeI = OPmoveM;

@@ -54,4 +58,9 @@
   OPmoveM.LocationMgrI -> LocationMgrC;
   OPmoveM.NeighborListI -> NeighborListProxy;
   OPmoveM.ErrorMgrI -> ErrorMgrProxy;
+  OPmoveM.MAC -> CBCMAC.MAC;
+
+  CBCMAC.BlockCipher -> Cipher.BlockCipher;
+  CBCMAC.BlockCipherInfo -> Cipher.BlockCipherInfo;
+  OPmoveM.CodeMgrI -> CodeMgrC;
 }
diff -Naur Agilla-orig/opcodes/OPmoveM.nc Agilla-ours/opcodes/OPmoveM.nc
--- Agilla-orig/opcodes/OPmoveM.nc 2006-12-14 12:28:16.929452800 -0600
+++ Agilla-ours/opcodes/OPmoveM.nc 2006-12-14 18:46:47.425542400 -0600
@@ -36,6 +36,9 @@
  * be found here: http://www.gnu.org/copyleft/lesser.html
  */

+#define AGILLA_MAC_KEY AC68BA4195BE3DEE8B9D692C9619E563
+
+
 includes Agilla;
 includes AgillaOpcodes;
```

```
@@ -52,14 +55,35 @@
     interface LocationMgrI;
     interface NeighborListI;
     interface ErrorMgrI;
+    interface MAC;
+    interface CodeMgrI;
   }
 }
 implementation {

   command result_t BytecodeI.execute(uint8_t instr, AgillaAgentContext* context) {
-    AgillaVariable arg;
+    AgillaVariable arg,macvar;
     uint16_t dest;
-
+
+     uint8_t key[16] = {0xAC, 0x68, 0xBA, 0x41, 0x95, 0xBE, 0x3D, 0xEE, 0x8B, 0x9D, 0x69,
+       0x2C, 0x96, 0x19, 0xE5, 0x63};
+     uint8_t codeSize,keySize = 16;
+     MACContext maccontext;
+     uint16_t i=0,k=0,j=0;
+     uint8_t codeCtr = 0;
+     uint8_t mac[4] = {0,0,0,0};
+     uint8_t mac1[4] = {0,0,0,0};
+     uint8_t mac2[4] = {0,0,0,0};
+     uint8_t mac3[4] = {0,0,0,0};
+     uint16_t* mac16 = 0;
+     uint8_t data[4] = {1,2,3,4};
+     uint8_t firstHashSize, secondHashSize;
+     uint8_t instruction;
+     uint8_t codeBlock[AGILLA_CODE_BLOCK_SIZE];
+     uint8_t done = 0;
+     uint8_t tmpmac[4] = {0,0,0,0};
+     result_t result = 0;
+
     // Get the destination address.
     if (call OpStackI.popOperand(context, &arg)) {
       if (arg.vtype & AGILLA_VAR_V)
@@ -71,6 +95,71 @@
         call ErrorMgrI.error(context, AGILLA_ERROR_TYPE_CHECK);
         return FAIL;
       }
+
+/* ************   Bobby and Tony's masterful additions *********** */
+
+    if (instr == IOPasmove || instr == IOPawmove) {
+
+     /*** prepare to be bedazzled ***/
+     call MAC.init(&maccontext, keySize, key);
+
+     // set condition code to 1, this is a hack
+     context->condition = 1;
+
+     if (instr == IOPawmove) {
+ // reset stack point and program counter
+ // on weak move.  This is also hack
+ context->pc = 0;
+ context->opStack.sp = 0;
+ instr = IOPasmove;
+     }
+
```

```
+     // debug the context
+        dbg(DBG_USR1, "Agent ID: %i\n", context->id.id);
+        dbg(DBG_USR1, "Agent PC: %i\n", context->pc);
+        dbg(DBG_USR1, "Agent CodeSize: %i\n", context->codeSize);
+        dbg(DBG_USR1, "Agent Condition: %i\n", context->condition);
+        dbg(DBG_USR1, "Agent instructions:\n");
+
+
+     firstHashSize = (context->opStack.sp)-1 + 8;
+     secondHashSize = 102;
+     call MAC.MAC(&maccontext, (uint8_t*)&(context->id),firstHashSize, mac1, 4);
+     call MAC.MAC(&maccontext, (uint8_t*)&(context->heap), secondHashSize, mac2, 4);
+
+/* *********  Calculate Code MAC ********** */
+
+  codeSize = context->codeSize;
+  while (!done) {
+    for (k=0; k<AGILLA_CODE_BLOCK_SIZE; k++) {
+      if (codeCtr < codeSize) {
+        instruction = call CodeMgrI.getInstruction(context, codeCtr);
+        codeCtr++;
+      } else {
+        done = 1;
+        instruction = 0;
+      }
+      codeBlock[k] = instruction;
+    }
+    // code block written, calculate the mac.
+    call MAC.MAC(&maccontext, codeBlock, AGILLA_CODE_BLOCK_SIZE, tmpmac, 4);
+    for(j=0; j<4; j++) {
+      mac3[j] = mac3[j] ^ tmpmac[j];
+    }
+  }
+
+
+     for(k=0; k<4; k++) {
+        mac[k] = mac1[k] ^ mac2[k] ^ mac3[k];
+     }
+     macvar.vtype = AGILLA_TYPE_READING;
+     mac16 = mac;
+     macvar.reading.type = *mac16;
+     mac16 = &mac[2];
+     macvar.reading.reading = *mac16;
+     call OpStackI.pushOperand(context, &macvar);
+   }
+


       // print debug statements
       switch(instr) {
@@ -112,8 +201,13 @@
           forward = call NeighborListI.getClosestNeighbor(&oneHopDest);
         #endif

-        if (forward)
-          return call AgentMgrI.migrate(context, oneHopDest, dest, instr);
+        if (forward) {
+
+          result = call AgentMgrI.migrate(context, oneHopDest, dest, instr);
+          if (result != SUCCESS)
+            context->condition = 0;
+          return result;
+        }
         else {
           context->condition = 0;
           return SUCCESS;
```

```
diff -Naur Agilla-orig/types/Agilla.h Agilla-ours/types/Agilla.h
--- Agilla-orig/types/Agilla.h 2006-12-14 12:28:17.840763200 -0600
+++ Agilla-ours/types/Agilla.h 2006-12-05 23:31:58.000000000 -0600
@@ -230,7 +230,7 @@
   AGILLA_TYPE_LOCATION = 64,
   AGILLA_TYPE_ANY      = AGILLA_TYPE_VALUE | AGILLA_TYPE_READING |
                          AGILLA_TYPE_STRING | AGILLA_TYPE_TYPE |
-                         AGILLA_TYPE_STYPE | AGILLA_TYPE_AGENTID | AGILLA_TYPE_LOCATION,
+                         AGILLA_TYPE_STYPE | AGILLA_TYPE_AGENTID | AGILLA_TYPE_LOCATION
 } AgillaDataType;

 typedef enum {
```

C. CBCMAC.NC PATCH FOR LARGE INPUTS

```
--- CBCMAC.nc.orig 2006-12-15 00:41:53.972852800 -0600
+++ CBCMAC.nc 2006-12-15 00:47:17.437972800 -0600
@@ -143,7 +143,7 @@
   async command result_t MAC.incrementalMAC (MACContext * context, uint8_t * msg,
       uint16_t msgLen)
     {
-      uint8_t i, pos = ((CBCMACContext*) context->context)->blockPos;
+      uint16_t i, pos = ((CBCMACContext*) context->context)->blockPos;
       uint8_t * partial = ((CBCMACContext*) context->context)->partial;

       // only proceed if the we're expecting less than msgLen of data.
```