View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Mecanismo para evitar ataques por confabulación basados en code passing

Marc Jaimez

Universitat Politecnica de Catalunya. C/ Jordi Girona 1 i 3. Campus Nord UPC.

Abstract. Los agentes móviles son entidades software formadas por código, datos, itinerario y estado, que pueden migrar de host en host autónomamente ejecutando su código. A pesar de sus ventajas, los aspectos de seguridad restringen enormemente el uso de código móvil. La protección del agente ante ataques de hosts maliciosos, es el problema de seguridad más difícil de resolver en los sistemas de agentes móviles. En particular, los ataques por confabulación han sido poco estudiados en la literatura. Este paper presenta un mecanismo de protección ante ataques por confabulación basados en code passing. Nuestra propuesta es un Multi-Code Agent que contiene diferentes variantes del código para cada host. Una Trusted Third Party es la responsable de proporcionar la información para extraer cada variante, y de tomar referencias temporales que se usarán para verificar la coherencia temporal.

Keywords: Mobile agent security, malicious hosts, collusion attack, code passing

1 Introducción

Los agentes móviles son entidades software que mueven código y datos a hosts remotos. Los agentes móviles pueden migrar de host en host realizando acciones de forma autónoma o en nombre de un usuario. El uso de agentes móviles permite ahorrar ancho banda, y permite una ejecución autónoma y off-line. Los agentes móviles son especialmente útiles para llevar a cabo tareas automáticamente, en casi todos los servicios electrónicos como el comercio electrónico y la administración de redes. A pesar de sus ventajas, el uso masivo de los agentes móviles se ve restringido por problemas de seguridad [12, 22]. Dentro de este escenario, se consideran dos entidades principales a la hora de estudiar los problemas de seguridad: el agente móvil, y el host que lo ejecuta. Éstos son los principales ataques: (1) el agente ataca al host: la protección del host se consigue usando técnicas de sand-boxing y un apropiado control de acceso [9]; (2) ataque a las comunicaciones: la protección del agente mientras está migrando de host en host se puede asegurar mediante el uso de los protocolos criptográficos conocidos [15]; y (3) el host ataca al agente: no existe por el momento ninguna solución publicada que proteja completamente a los agentes ante estos ataques. Este último ataque es conocido como el problema de los hosts maliciosos.

Este paper introduce un nuevo mecanismo de defensa ante ataques por confabulación basados en code passing. En este tipo de ataque, un host envía el código del agente a otro host que no es el siguiente destinatario de la ruta, y de esta forma este host puede analizar el código del agente antes que llegue a través de la ruta legal. Los ataques por confabulación son difíciles de prevenir o detectar, y es por esta razón que la mayoría de propuestas no los resisten. Solo algunas propuestas intentan limitar estos ataques usando técnicas de protección del itinerario [6, 3, 19]. El uso de nuestro mecanismo implica la ejecución de un agente distinto en cada host. Para hacerlo, generamos distintas variantes del código del agente, y las unimos para construir un *Multi-Code Agent* (MCA), que será la entidad final que se enviará a los hosts. Cuando un hosts reciba el MCA, necesitara cierta información (*extracting instructions*) para poder extraer la variante del agente que le corresponda, estas *extracting instructions*serán solicitadas a una Trusted Third Party (TTP). Éstas peticiones, serán usadas también por la TTP para tomar referencias temporales que servirán para controlar los tiempos de ejecución de los hosts, permitiendo detectar comportamientos maliciosos. El MCA se construye de forma que se pueda evitar que dos o más hosts confabulen para analizar sus variantes y localizar vulnerabilidades en el código.

2 Hosts Maliciosos

Los ataques realizados por un host malicioso que está ejecutando un agente móvil son, con diferencia, el problema de seguridad más difícil de resolver en un sistema de agentes móviles [8, 13]. Aunque es posible proteger la migración del agente móvil de escuchas o manipulaciones, mediante el uso de la firma digital o técnicas criptográficas (communications security), es difícil detectar o prevenir ataques realizados por hosts maliciosos durante la ejecución del agente (por ejemplo proporcionar integridad y privacidad). Los hosts maliciosos pueden intentar sacar provecho del agente, si leen o modifican el código, los datos, el flujo de ejecución, las comunicaciones o incluso los resultados. El agente, por supuesto, no puede transportar una clave de descifrado ya que el host podría leerla [5]. Asimismo, si dos o más hosts maliciosos colaboran para realizar un ataque, la tarea de proteger el agente se torna aún más difícil.

En este paper asumimos que los hosts no son entidades de confianza (pueden intentar atacar un agente cuando lo estén ejecutando). Las propuestas publicadas que intentan conseguir integridad de ejecución y privacidad, se pueden dividir en dos categorías principales: propuestas de detección de ataques y propuestas de prevención de ataques. Por un lado, las propuestas de prevención de ataques intentar evitar ataques de eavesdropping o manipulación, antes de que ocurran. Por otro lado, el objetivo de las propuestas de detección de ataques consiste en prevenir los ataques disuadiendo a los hosts maliciosos.

Obviamente, prevenir es mejor que curar, por lo que las propuestas de prevención son más eficientes en términos de seguridad. Sin embargo, según nuestra opinión las técnicas de prevención de ataques son difíciles de implementar, o sus costes computacionales las hacen difíciles de usar en escenarios reales. De echo,

no existe ninguna propuesta de prevención de ataques que proporcione una resistencia satisfactoria. Por otra parte, las técnicas de detección de ataques suelen ser más fáciles de implementar, y por ello las consideramos más prometedoras. No obstante, las técnicas de detección de ataques son efectivas solo en aquellos escenarios en los que exista la posibilidad de penalizar a los hosts maliciosos, y en los que las penalizaciones sean mayores que los beneficios obtenidos por el atacante.

2.1 Propuestas de prevencion de ataques

Aquí resumimos las principales propuestas que intentan evitar ataques de eavesdropping y manipulaciones, proporcionando privacidad e integridad en la ejecución.

En [23], Ordille propone ejecutar los agentes solo en maquinas de confianza, esto es, máquinas de las cuales no se espera ningún tipo de actividad anómala o maliciosa. Sin embargo, esta propuesta no es útil en una red abierta como Internet porque existen pocos hosts de confianza. Se puede pensar, de forma más general, en un cierto control social y establecer unos varemos de reputación de las plataformas que ejecutan los agentes. A partir de estas relaciones de confianza entre entidades es posible inferir otras relaciones que pueden ser usadas para determinar el grado de confianza de un host [14, 20]. Desafortunadamente, estas propuestas no describen los mecanismos de seguridad a utilizar una vez que se ha determinado el nivel de confianza. Algunos autores proponen el uso de un subsistema cerrado donde se ejecutan de forma segura los agentes y al cual no tiene acceso ni el propio dueño de la plataforma [31, 30, 17]. El principal inconveniente de esta propuesta, es que obligaría a cada host con capacidad para ejecutar agentes a adquirir un equipo hardware. Además, es dudosa la confianza que se puede depositar en el hardware de un determinado suministrador.

En [26], Roth presenta la idea de la protección mutua. En un entorno abierto como Internet, se puede asumir que las relaciones de confianza están limitadas, y por tanto la confabulación entre hosts es poco probable. Por esta razón, el agente móvil se envía junto con otros agentes cooperativos a través de itinerarios disjuntos. El almacenamiento de datos confidenciales y la toma de decisiones del agente móvil se realizan en esos agentes cooperativos, y por tanto los ataques por eavesdropping o manipulaciones no se pueden realizar directamente sobre el agente móvil, sino que deben ser efectuados sobre los agentes cooperativos. Por desgracia, el sistema debe garantizar que las comunicaciones entre los agentes cooperativos sean posibles durante toda la transacción. Adicionalmente, tienen que haber mecanismos para recuperar los resultados en caso que un agente se pierda. Todo y con esto, la posibilidad de confabulación no desaparece por completo.

En [21], se define la entropía de agentes móviles como medida métrica para calcular las intenciones del agente. La idea de esta propuesta es hacer que las intenciones del agente sean desconocidas para el host, porque un host malicioso que no sea capaz de interpretar las intenciones del agente móvil, no será capaz de leer o modificar el agente para sacar beneficio. Esto se puede hacer mediante (1) *intention spreading*, disminuyendola entropía haciendo que el agente

ejecute tareas que el usuario no ha pedido; o mediante (2) *intention shrinking*, aumentando la entropía mediante la distribución de las intenciones del usuario en varios agentes cooperativos. En el primer caso, nada evita que los hosts maliciosos obtengan beneficios de todas las tareas. En el segundo caso, tenemos los mismos problemas que en la propuesta previa de protección mutua [26].

La Generación de Claves Dependientes del Entorno presentada en [25] hace que el código del agente sea imposible de descifrar hasta que se den las condiciones propicias, así el análisis previo por parte del host se evita por completo. No obstante, el principal problema de esta propuesta es que el agente es vulnerable una vez ha sido descifrado, y por tanto, la privacidad y la integridad de ejecución no se pueden asegurar. Además, la propuesta obliga a mantener una continua monitorización del entorno.

Una Blackbox es un entorno software que solo permite la lectura de las entradas y las salidas, y los datos internos no pueden ser leídos ni modificados. Desafortunadamente, no hay ningún algoritmo conocido con estas propiedades. La Blackbox limitada en tiempo [10] tiene este nivel de seguridad pero solo durante un periodo de tiempo limitado. Transcurrido este tiempo, ni la privacidad ni la integridad de la ejecución se pueden garantizar. El mecanismo de Hohl se basa en un algoritmo de mess-up que ofusca el código y los datos para hacerlos difíciles de entender, y por ende de modificar. La principal dificultad en este caso es como estimar el tiempo durante el cual la ejecución es segura. Por otrolado, tampoco existen mecanismos para evaluar la bondad de un algoritmo de mess-up. Adicionalmente, un host malicioso con suficiente tiempo y recursos podría analizar el agente ofuscado para sacar información, o manipular el agente para obtener una ejecución favorable.

El uso de programas cifrados [27] se propone como la única manera de proporcionar privacidad y integridad de ejecución al código móvil. Los hosts ejecutan el código cifrado directamente. La función para descifrar, es utilizada para recuperar los resultados cuando el agente llega al host origen. En [4], la propuesta se mejora de forma que los agentes pueden atravesar múltiples hosts. En [1], el esquema propuesto permite a los agentes tomar decisiones mientras viajan, gracias al uso de una TTP. La dificultad con la que se encuentran estas propuestas, es encontrar funciones que pueden ser ejecutadas de forma cifrada.

2.2 Propuestas de Detección de Ataques

Aquí resumimos las principales propuestas publicadas que intentan detectar manipulaciones para proporcionar integridad de ejecución.

En [18], se introduce la idea de replicación y voto. En cada etapa, los hosts ejecutan el agente en paralelo y envían varias réplicas del agente a un conjunto independiente de hosts en la siguiente etapa. En algunas de las etapas, los hosts comparan los resultados de los agentes y escogen los resultados correctos por mayoría. Esta propuesta no solo proporciona un mecanismo de tolerancia a fallos, sino que detecta aquellos hosts que han realizado un ataque de manipulación. Se asume que los resultados de todas las réplicas deben ser los mismos si todos los hosts han actuado honestamente, por tanto todos los hosts en una misma etapa

tienen que tener los mismos recursos y datos. Desafortunadamente, esto no es coherente con la propiedad de independencia de los host, por ejemplo los hosts pueden tener distintos intereses para atacar al agente.

En [29, 28], Vigna introduce la idea de las trazas criptográficas, que son logs de las operaciones realizadas por el agente durante su ejecución. Con dichas trazas, se puede volver a ejecutar el agente para verificar su ejecución en el host. Si el host origen sospecha que un host ha modificado el agente y quiere verificarlo, pide las trazas y vuelve a ejecutar el agente. Si la nueva ejecución no coincide con las trazas, el host nos esta intentando engañar. En lugar de las trazas, el host envía un hash de estas para evitar ataques de repudio. La propuesta no solo detecta manipulaciones, sino que también proporciona una prueba del comportamiento malicioso del host. Sin embargo, esta propuesta tiene dos inconvenientes: (1) la verificación solo se lleva a cabo en caso de sospecha, pero no se explica como se detecta a un host sospechoso; (2) durante un periodo indeterminado, cada host tiene que reservar suficiente capacidad para almacenar las trazas ya que el host origen se las puede pedir. [7], aun así, consideramos que el uso de trazas sigue siendo muy costoso para todas las entidades involucradas.

En la propuesta de los Estados de Referencia[1][11], la correcta ejecución en un host se verifica en el siguiente host. Esta propuesta se basa en las trazas criptográficas de Vigna, pero tiene el problema de enviar los datos de entrada del agente (que pueden ser confidenciales) al siguiente host para verificar la ejecución. Algunas propuestas posteriores están basados también en las trazas de Vigna [32, 16], pero ninguna de ellas soluciona el problema de detectar manipulaciones de forma satisfactoria.

En [2] se usa un agente clon para realizar una ejecución de referencia del agente. Si un host ha alterado el código del agente o la ejecución, podemos compararla con la obtenida por el clon de referencia. Con esta propuesta podemos detectar los ataques prácticamente en el momento de llevarse a cabo. Sin embargo, las continuas verificaciones de la ejecución implican una constante comunicación entre agentes. Además, la ejecución de dos agentes (uno ejecutado por el host, y el otro en una TTP) conlleva un gasto mayor de ancho de banda y recursos computacionales. En [24], distintos agentes (agentes móviles y agentes sedentarios) cooperan, el agente móvil es ejecutado por los hosts y los agentes sedentarios son ejecutados en plataformas de confianza que generan ejecuciones de referencia.

3 Multi-Code Agent

En este paper presentamos un nuevo mecanismo que permite resistir ataques por confabulación basados en Code Passing. Tal y como hemos mencionado previamente, los ataques por confabulación en sistemas de agentes móviles apenas han sido estudiados. En un ataque basado en Code Passing, un host envía el código del agente a otro host que no es el siguiente destinatario en la ruta del agente,

[1] Los estados de referencia son aquellos que han sido producidos en hosts de referencia (de confianza)

sino que es cualquier otro dentro del itinerario; gracias a esto, este host puede disponer del código del agente mucho antes de recibirlo a través de la ruta legal, y por tanto puede aprovechar ese tiempo para analizarlo en busca de posibles vulnerabilidades.

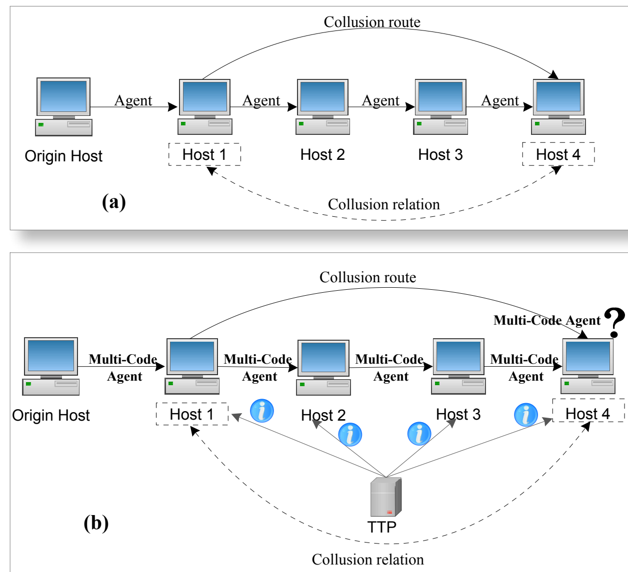

Fig. 1. El problema del Code Passing

En la Figura 1.a se muestra un ataque por confabulación basado en Code Passing: el host origen envía el agente móvil al Host-1, que ejecuta el agente de forma normal y lo envía al próximo host (Host-2) a través del itinerario marcado. Al mismo tiempo, el Host-1 envía el código del agente a otro host dentro del itinerario, llamémosle Host-4, quien dispondrá de más tiempo para analizar el código del agente en busca de posibles vulnerabilidades.

Para evitarlo, generaremos diferentes *variantes* del código del agente original, las uniremos de forma adecuada, y construiremos un Multi-Code Agent (MCA), que será la entidad final que se enviará a todos los hosts. Cuando un hosts reciba el MCA, necesitara cierta información (*extracting instructions*) para poder extraer la variante del agente que le corresponda, estas *extracting instructions* serán solicitadas a una Trusted Third Party (TTP). A su vez, la TTP se encargará de obtener referencias temporales para controlar los tiempos de ejecución de los host. Finalmente, cuando el host finalice la ejecución de su variante, enviará el MCA al siguiente host y el proceso empezará de nuevo. El MCA se diseña por tanto con el objetivo de evitar que dos o mas hosts confabulen

y puedan analizar el código de sus variantes para hallar vulnerabilidades. Las referencias temporales se usan para limitar el tiempo de ejecución en cada host, con lo que se pueden detectar comportamientos maliciosos.

En la Figura 1.b se muestra la idea principal de este mecanismo. El host origen envía el MCA al primer host del itinerario, Host-1. El Host-1 solicita las *extracting instructions* a la TTP para poder extraer su correspondiente variante, y a su vez la TTP usa esta solicitud para tomar una referencia temporal de confianza. Seguidamente, el Host-1 ejecuta su variante y envía el MCA al siguiente host en el itinerario, Host-2. El Host-1 sigue teniendo la posibilidad de enviar el MCA (o incluso su propia variante) a otro host directamente, por ejemplo al Host-4, a través de la ruta de confabulación. Sin embargo, este último no podrá extraer su variante sin disponer de las *extracting instructions* pertinentes. Además, como la TTP controla los tiempos de ejecución, el Host-4 no podrá solicitar las *extracting instructions* hasta que el Host-3 le envíe el MCA.

3.1 Etapa de Codificación

La etapa de codificación empieza con la modificación del código del agente original para obtener las diferentes variantes (ver Figura 2.a). Tal como se muestra, el agente original tiene una sección de Código, que contiene todo el Bytecode del agente, y una sección de Resultados que hace referencia a la estructura de datos dónde se guardan los resultados de la ejecución. En este ejemplo en particular hay tres variantes del agente original, ya que el agente se va a ejecutar en tres hosts distintos. Estas tres nuevas variantes del agente deben cumplir dos funciones básicas: proporcionar Bytecodes distintos, y proporcionar una estructura de datos distinta para los resultados de ejecución de cada uno de los host. Después de obtener todas las variantes, el codificador toma el Bytecode de cada una de ellas y construye el MCA. Este proceso se divide en tres fases: la fase de *Merging*, la fase de *Meshing up*, y la fase de *Compressing*. En la fase de *Merging* (ver Figura 2.b) se toma el Bytecode de todas y cada una de las variantes las variante y se pone todo junto en un único archivo. En la fase de *Meshing up* (ver Figura 2.c) se mezclan las instrucciones Bytecode de los distintas variantes. Finalmente, en la fase de *Compressing* (ver Figura2.d) se elimina redundancia de código. Al final de la fase de codificación, tenemos un MCA que tiene una sección de Código (que contiene el Bytecode de todas las variantes) y una sección de Resultados dónde se almacenan los resultados específicos de cada host.

Creación de los nuevos agentes El Multi-Code Agent contiene el código de todas las variantes del agente original, y estas deben cumplir dos funciones básicas: proporcionar distintos Bytecodes (queremos que cada host ejecute un agente distinto), y proporcionar distintas estructuras de datos para los resultados de ejecución (queremos verificar que cada host ha ejecutado el agente correcto). El proceso de creación de las variantes del agente original se divide en dos pasos:

Paso1. Modificación del código para obtener diferentes estructuras de datos En este paso, modificamos el código del agente original para que cada una de las

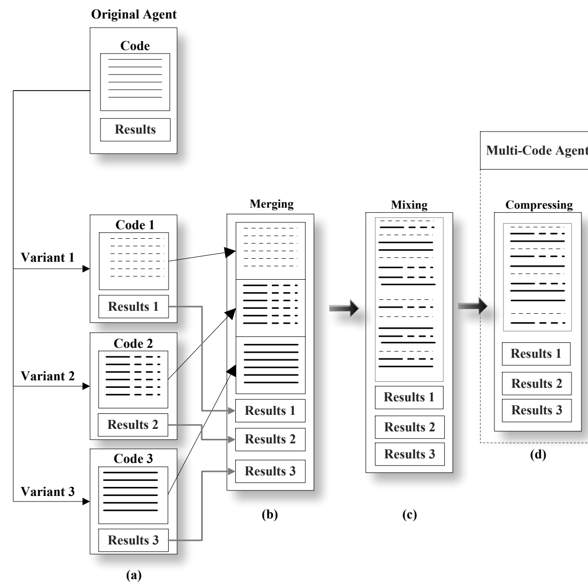

Fig. 2. Etapa de codificacion

variantes genere una sección de Resultados distinta. Cada una de estas secciones de Resultados contendrá una *Identity Mark*, que servirá para comprobar que cada host ha ejecutado su correspondiente variante. La forma en que cada host dispondrá toda esta información (los propios valores, su orden y sus relaciones) en la sección de Resultados definirá como implementa su *Identity Mark*. La sección de Resultados, finalmente se adjuntará al MCA para ser enviado al siguiente host.

Paso2. Modificación del código para obtener variabilidad en el Bytecode El objetivo de este paso es obtener un conjunto de nuevas variantes, distintas y difíciles de analizar, que realicen las mismas tareas que el agente original pero usando un código distinto. Sin embargo, recomendamos que estas nuevas variantes tengan una estructura similar (mismas clases, métodos, nombres, variables, etc) y solo se modifique el código de los métodos para introducir diferencias entre variantes. Si respetamos esta restricción, seremos capaces de reducir significativamente la longitud del MCA, la longitud de las *extracting instructions*, y la complejidad de las *extracting instructions*. Es importante recalcar que cuando decimos que todas las variantes deben tener una estructura similar, esto no significa que se deba mantener la estructura del agente original.

Para ilustrar la importancia de mantener la estructura básica de las clase en todas las variantes, mostraremos un ejemplo con un agente original y dos variantes. La Figura 3.a muestra el código del agente original, que esta compuesto por una única clase llamada OriginalAgentCode. Esta clase contiene un único método llamado method1(), que realiza un simple cálculo aritmético $a = 10+20$,

y devuelve el resultado. Como se puede ver, para crear las nuevas variantes mantenemos la misma estructura del agente original (una clase y un método) pero añadimos dos nuevas variables, b y c. Estas dos nuevas variables nos permitirán realizar las mismas operaciones que realizaba el agente original, pero de manera distinta. La Figura 3.b muestra el código de la primera variante, y la Figura 3.c muestra el código de la segunda variante. La única diferencia entre ambas variantes es el orden en que se usan las variables.

```
public class OriginalAgentCode {

   int method1( ) {

      int a = 10 + 20;
      return a;
   }
}
```

(a)

```
public class OriginalAgentCodeA {

   int method1( ) {

      int a;
      int b;
      int c;

      a = 10;
      b = 10 + a;
      c = a + b;
      return c;
   }
}
```

(b)

```
public class OriginalAgentCodeB {

   int method1( ) {

      int a;
      int b;
      int c;

      c = 10;
      a = c + 10;
      b = a + c;
      return b;
   }
}
```

(c)

Fig. 3. Código original y código de las nuevas variantes

Si analizamos el código del ejemplo, ambas variantes tienen una longitud de 3328 bits, pero tan solo 144 bits de los 3328 son distintos para cada variante. Esto significa que solo un 3,42% del Bytecode difiere de una variante a otra, y es debido al hecho que hemos usado la misma estructura para las dos variantes. Si hubieran más de dos hosts en el itinerario, necesitaríamos mas variantes del agente original, y por tanto deberíamos implementar la misma operación aritmética de manera distinta para cada nueva variante. Para hacerlo, simplemente deberíamos incluir nuevas variables y cambiar el orden en que las usamos.

Construyendo una Identity Mark La *Identity Mark* es un vector que contiene n valores. Estos valores pueden ser resultados de ejecución R_i, o pueden ser valores intermedios V_j. Un ejemplo sencillo de una posible implementación de una *Identity Mark* con $n = 5$ se muestra en la Figura 4. El Host ejecuta el código de su variante y obtiene dos valores que se corresponden a los resultados de ejecución (R_1y R_2), y tres valores intermedios (V_1, V_2, V_3). Con todos estos valores, el agente construye una estructura de datos que implementa su *Identity Mark*, y finalmente la envía junto al MCA.

La fortaleza de la *Identity Mark* se basa en dos características principales:

1. El orden de los valores: la *Identity Mark* es un vector ordenado de n valores, por tanto, existen $n!$ *Identity Marks* distintas que se pueden obtener simplemente ordenando los valores de manera distinta. De esta manera, la probabilidad de generar la *Identity Mark* de una cierta variante es de $\frac{1}{n!}$. En

el ejemplo de la Figura 4, la *Identity Mark* contiene 5 valores (R_1, R_2, V_1, V_2, V_3), que pueden generar ciento veinte secuencias distintas.
2. Los valores intermedios: no solo el orden de los valores es específico de cada agente, sino que los propios valores se obtienen de forma distinta dependiendo de la variante. Nosotros asumimos que los valores correspondientes a los resultados de ejecución (R_i) no pueden ser cambiados. Sin embargo, los valores intermedios V_j se pueden calcular de distintas formas segun la variante. Por ejemplo, en el ejemplo de la Figura 4, el cálculo de V_2 en el Agente A podría depender de R_1, V_1 o cualquier otro dato de entrada usado solo por el Agente A. Por otro lado, el cálculo de V_2 en el Agente B puede depender solo de R_2. Gracias a esto, un host no puede establecer ninguna relación directa entre el V_2 del Agente A, y el V_2 del Agente B.

Tal y como se puede deducir, si incrementamos el número de valores n del vector *Identity Mark,* aumentaremos las posibilidades de proteger el agente de ataques por confabulación.

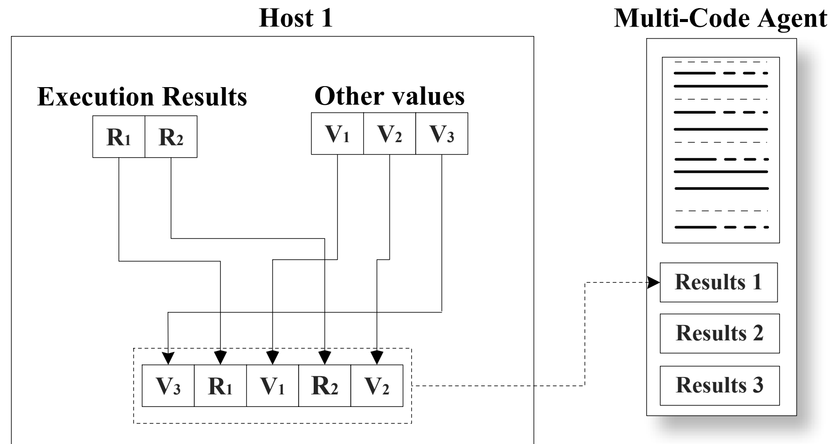

Fig. 4. Ejemplo de una Identity Mark

Merging En esta fase, compilamos los archivos .java, obtenemos los archivos .class, y los unimos para construir el MCA. Siguiendo con el ejemplo propuesto en la Subsección 3.1, compilamos los archivos OriginalAgentCodeA.java y OriginalAgentCodeB.java, y obtenemos dos nuevos archivos: OriginalAgentCodeA.class y OriginalAgentCodeB.class.

La Figura 5.a y la Figura 5.b nos muestran sendos diagramas representativos de la estructura de los archivos .class. Las celdas blancas representan el código compartido por ambas variantes, las celdas negras representan el código exclusivo

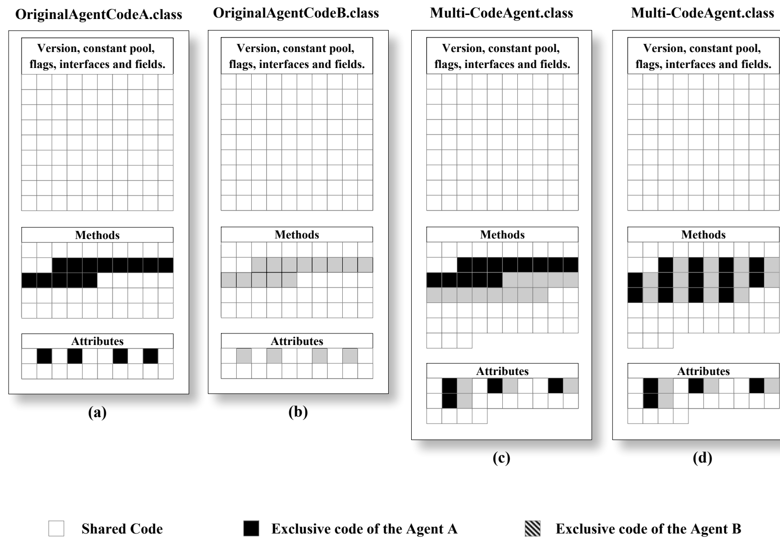

Fig. 5. Building up the Multi-Code agent

de la variante A, y las celdas grises representan el código exclusivo de la variante B. La Figura 5.c muestra como quedaría la estructura del archivo .class del Multi-Code Agent. Como podemos observar, hemos construido un único archivo .class que contiene dos implementaciones distintas del agente original, y que además tiene la mayor parte del código compartido por ambas variantes.

Un ejemplo detallado de la fase de Merging se da en la Figura 6. Partiendo del agente original (ver Figura 6.a), se crean las dos nuevas variantes. La Figura 6.b se corresponde con el Bytecode específico del method1() de la variante A (color negro). La Figura 6.c se corresponde con el Bytecode específico del method1() de la variante B (color gris). Finalmente, la Figura 6.d muestra el Bytecode resultante del MCA (sin haber pasado todavía por las fases de mixing y compressing). Por simplicidad, el resto del Bytecode no se muestra en la Figura 6.

Mixing Hemos partido de un agente original y lo hemos modificado para obtener dos implementaciones distintas del mismo. Seguidamente, estos dos nuevos agente han sido unidos para generar un MCA, y ahora vamos a mezclar su código. La idea básica de la fase de Mixing se muestra en la Figura 5.d. Antes de realizar el Mixing, las celdas negras y grises están separadas y por tanto podrían ser identificadas por un atacante. Después de realizar el Mixing, un atacante tendrá más dificultades para identificar las celdas de cada variante.

Un ejemplo detallado de la fase de Mixing se puede ver en la Figura 6.e, que nos muestra una posible distribución final de las instrucciones Bytecode después del Mixing. Es importante recalcar que el orden en que disponemos las instrucciones dentro del MCA podría ser cualquiera. De echo, cuanto más alter-

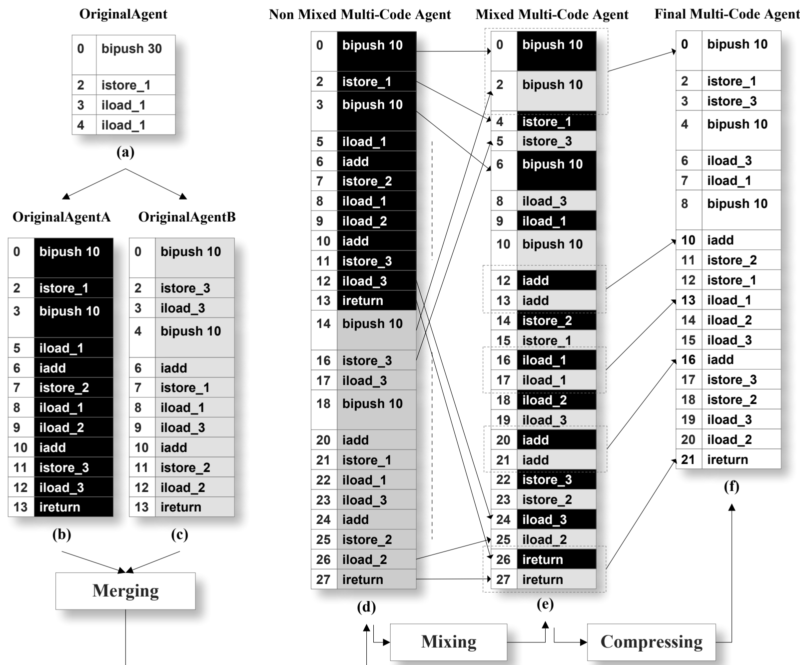

Fig. 6. A bytecode view of the Multi-Code Agent construction

emos dicho orden, más difícil y complicado le resultará a un atacante entender el código. Sin embargo, hay que tener en cuenta que la complejidad y la longitud de las *extracting instructions* también se verán incrementadas. Por lo tanto, en nuestro caso preservaremos el orden natural de las instrucciones para simplificar las *extracting instructions* .

Compressing La última fase dentro de la etapa de Codificación del MCA, consiste en comprimir el conjunto de instrucciones Bytecode específicas de cada variante. Como resultado de la fase de Mixing, algunas instrucciones consecutivas son idénticas, y cada vez que se de este echo podemos eliminar una de estas instrucciones (ver Figura 6.f). Aunque la longitud del código se reduzca al eliminar algunas de las instrucciones, el factor de compresión es muy bajo. En el caso del ejemplo, la longitud del MCA antes de la compresión es de 3472 bits, y la longitud después de la compresión es de 3424 bits. Esto significa que hemos reducido la longitud del MCA en un 1.38%, que es prácticamente despreciable. Todo y con esto, la fase de compresión es necesaria para añadir complejidad al código resultante. Para un atacante resultará más difícil extraer una variante si algunas instrucciones han sido borradas.

Extracting instructions Cuando un host recibe el MCA, debe extraer su correspondiente variante, y para hacerlo necesita saber que instrucciones pertenecen a su propio agente. Esta información está contenida en lo que llamamos las *extracting instructions*, que no son más que una lista de posiciones de memoria. Estas posiciones de memoria indican que instrucciones Bytecode no pertenecen a nuestra variante (lista negra). La Figura 7 nos muestra el proceso de generación de las *extracting instructions*. El MCA contiene una mezcla comprimida del código de las distintas variantes (ver Figura 7.a). Las celdas negras contienen instrucciones del agente A; las celdas grises contienen instrucciones del agente B; y las celdas blancas contienen instrucciones compartidas por ambos agentes. Por ejemplo, para obtener el agente A, solo necesitamos preservar aquellas celdas que contienen instrucciones del agente A (ver Figura 7.b). Lo mismo ocurre para extraer el agente B (ver Figura 7.c). Una vez tenemos identificadas las celdas que contienen las instrucciones de otros agentes, guardamos esta información en un vector (ver Figura 7.d, y Figura 7.e) y lo mandamos a la TTP .

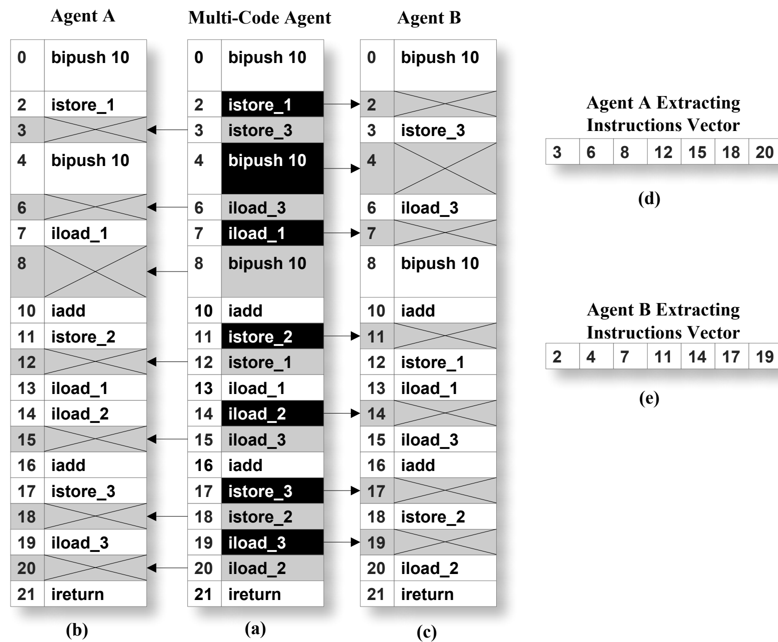

Fig. 7. Generación de las Extracting Instructions

Si generalizamos el esquema anterior para un itinerario formado por un número más elevado de hosts, vemos que debemos adaptar la manera de codificar las *extracting instructions*. En el ejemplo anterior, hemos codificado las *extracting instructions* como una blacklist, dónde simplemente indicamos aquel-

las instrucciones que no pertenecen nuestra variante. A medida que aumentamos el número de hosts (y por tanto el número de variantes), cada vez hay más instrucciones pertenecientes a otras variantes. Llegados a este punto no es óptimo indicar una por una las instrucciones a desechar, ya que la longitud de las *extracting instructions* se multiplicaría por un factor igual al número de variantes del agente. La solución final que adoptamos consiste en especificar rangos de instrucciones en lugar de instrucciones concretas. En la fase de Mixing, realizamos un entrelazado con las instrucciones específicas de cada variante, y de esta forma se generan conjuntos de instrucciones que pertenecen a distintas implementaciones, pero que a su vez hacen referencia al mismo segmento de código en cada una de variantes. En la Figura 8.a se muestra parte del código entrelazado de un MCA formado por 16 variantes distintas del agente original. En concreto este código entrelazado pertenece a la sección de código de uno de los métodos de los agentes. Si observamos la primera posición de memoria (@202), vemos que alberga la primera instrucción Bytecode de la primera variante, la posición siguiente (@203) alberga la segunda instrucción Bytecode de la segunda variante, y así sucesivamente hasta la posición @217 que contiene la primera instrucción de la última variante. Si queremos generar las *extracting instructions* para la variante 4 por ejemplo, debemos indicar los conjuntos de instrucciones que no pertenecen a esta variante: $[@202, @204] \cup [@206, @221] \cup \cdots \cup [@410, @427]$.

La estructura final de las *extracting instructions* se puede ver en la Figura 8.b. El primer bloque (resaltado en gris), está formado por las direcciones de inicio de las distintas secciones con código entrelazado, mientras que el segundo bloque (resaltado en negro), esta formado por las longitudes de cada conjunto de instrucciones. Si nos fijamos en el primer bloque, los primeros 16 bits de información se usan para indicar el total de secciones distintas que conforman el código del MCA. Como hemos explicado anteriormente, la mayor parte del Bytecode del MCA es compartido por todas las variantes. Sin embargo, el código referenete a las operaciones realizadas por los métodos es distinto para cada variante, y el código referenete a los atributos de estos métodos también difiere segun la variante (vease Figura 5). Una vez sabemos cuantas secciones con código entrelazado hay, proporcionamos las direcciones de inicio de cada una de ellas. El segundo bloque, nos da la información necesaria para detectar aquellos conjuntos de instrucciones contenidos en una misma sección que son deshechables o que pertenecen a nuestra variante. Para indicar el tamaño de los conjuntos de instrucciones a desechar usamos bloques de 5 bits, y para indicar el tamaño de los conjuntos de instrucciones a conservar usamos bloques de 3 bits. La elección del tamaño del bloque para las instrucciones de deshecho depende del número de variantes que vamos a usar, cuantas más variantes se utilicen más largos serán los conjuntos de (típicamente el tamaño será igual al número de variantes). En cambio los conjuntos de instrucciones a conservar siempre seran de longitud parecida (típicamente 1, 2 o 3 dependiendo de si hay alguna instrucción compartida) y no dependerán del número de variantes. Por último, para indicar que un cambio de sección pondremos todos los bits de un bloque de 5 bits a 0, y de este modo

sabremos que el siguiente bloque hace referencia a un conjunto de instrucciones de la siguiente sección

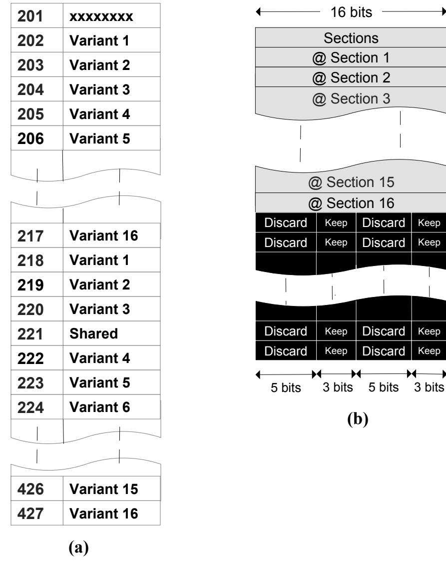

Fig. 8. Estructura final de las Extracting Instructions

3.2 Etapa de Distribución

Antes de enviar el MCA, el host origen tiene que enviar todas las *extracting instructions* a la TTP (un vector por cada host del itinerario). Ésta sen encargará de gestionar el envío de las *extracting instructions* a los hosts. Gracias a esto, el host origen queda liberado de esta tarea y no tiene que permanecer online mientras el MCA realiza su migración. Este proceso se muestra en la Figura 9:

- Paso 1: el host origen envía el conjunto de *extracting instructions* (I_1, I_2,..., I_N) a la TTP para su distribución (en el ejemplo $N = 2$).
- Paso 2: el host origen envía el MCA al primer host del itinerario (Host-1). Después de este paso, el host origen puede estar offline hasta que el MCA salga del último host con todos los resultados.
- Paso 3: el Host-1 recibe el MCA y realiza una petición de *extracting instructions* a la TTP, para poder extraer su variante.
- Paso 4: la TTP autentica la petición del Host-1, y le envía su *extracing instructions vector*, I_1. Además almacena una referencia temporal T_1 del instante en el que el Host-1 ha pedido el vector.

– Paso 5: el Host-1 extrae el código de su variante, la ejecuta, y adjunta los resultados de la ejecución D_1 al MCA antes de enviarlo al siguiente host (Host-2).
– Paso 6 y 7: el resto de hosts del itinerario realizan el mismo proceso que el Host-1.
– Paso 8: cuando el último host ha recibido sus *extracting instructions*, la TTP envía todas las referencias temporales recogidas (T_1, ..., T_N) al host origen .
– Paso 9: el último host envía el MCA con todos los resultados de ejecución (D_1, ... ,D_N) al host origen.

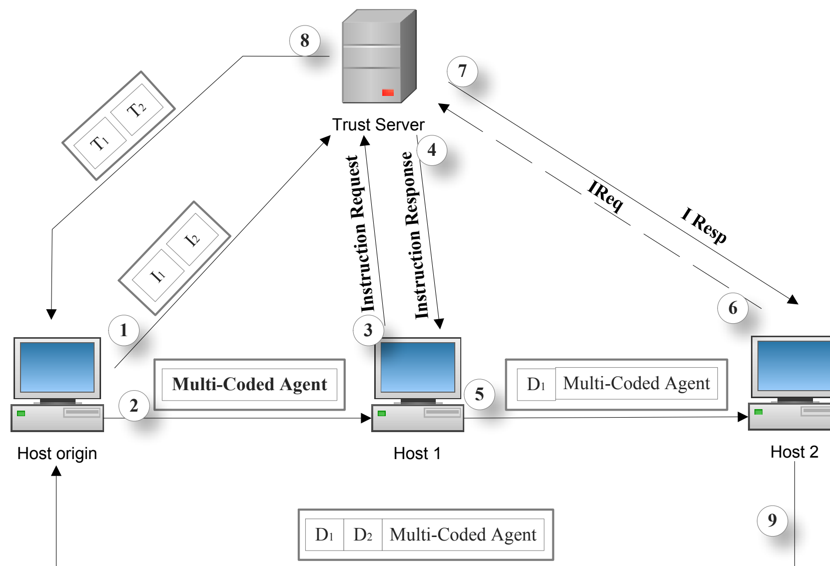

Fig. 9. Distribucion del Multi-Code Agent

3.3 Etapa de verificación de las referencias temporales

La TTP ha estado tomando referencias temporales cada vez que ha recibido una petición de *extracting Instructions* (IReq) válida. Gracias a esto, va a ser posible (a posteriori) calcular el periodo de tiempo en el que cada host ha dispuesto de su variante. La Figura 10 nos muestra un diagrama temporal en el que se puede apreciar que la TTP toma la referencia temporal justo en el momento en que envía el Instruction Response message (IResp) al host. Estas referencias temporales se utilizan en el host origen para verificar la coherencia de tiempos de ejecución en los hosts. El host origen calcula $\triangle T_i = T_{i+1} - T_i$, que incluye el tiempo de extracción del agente, el tiempo de ejecución de agente, el tiempo de

transmisión del MCA al siguiente host, el retardo de propagación, y el intervalo de tiempo desde que el host siguiente recibe el MCA hasta que este recibe el IResp. Con toda esta información, el host origen es capaz de estimar el tiempo de ejecución de cada agente.

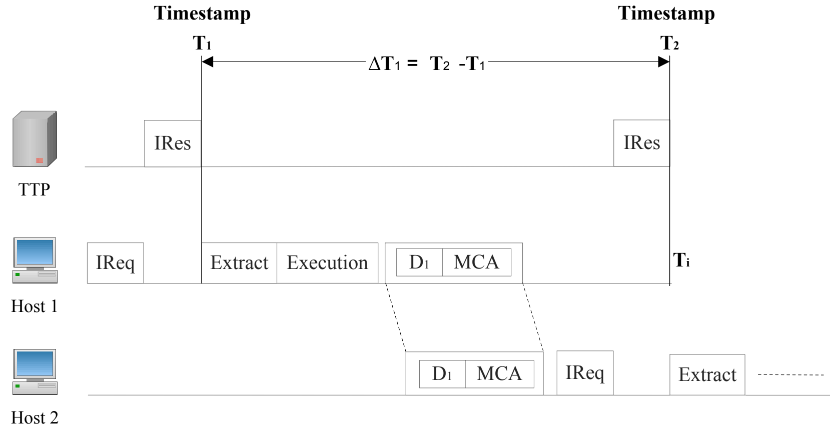

Fig. 10. Timestamps storing

Dependiendo de las características computacionales de los hosts, y de los recursos necesarios para ejecutar los agentes, el host origen estima un tiempo máximo de ejecución permitido ($\triangle T_{Max}$). En el caso que algún $\triangle T_i$ sea mayor que este valor, el host será etiquetado como sospechoso de realizar actividades maliciosas. La Figura 11 nos muestra dos lineas temporales, una en la que el Host-2 actúa honestamente, y otra en la que actúa maliciosamente. Si el Host-2 actúa honestamente, no rebasará el máximo tiempo de ejecución, y por tanto, el host origen detectará que $\triangle T_{HonestHost} < \triangle T_{Max}$. Por otro lado, si el Host-2 intenta analizar el código del agente en busca de vulnerabilidades, rebasará el tiempo máximo de ejecución, y el host origen detectará que $\triangle T_{MaliciousHost} > \triangle T_{Max}$. En este caso el Host-2 será sospechoso de actuar maliciosamente.

3.4 Fase de verificación de la coherencia de los datos

Como hemos mencionado con anterioridad, cada agente tiene una Identity Mark propia, que se codifica en los resultados de ejecución. Analizando estas Identity Marks, seremos capaces de verificar que variante ha sido ejecutada por cada host. Concretamente, al final del proceso de distribución, el host origen recibe el MCA con los resultados de ejecución, y una por una va extrayendo todas las Identity Marks. Si al extraer una Identity Mark, comprovamos que se corresponde con la que esperamos obtener, entonces tenemos un resultado positivo y asumimos

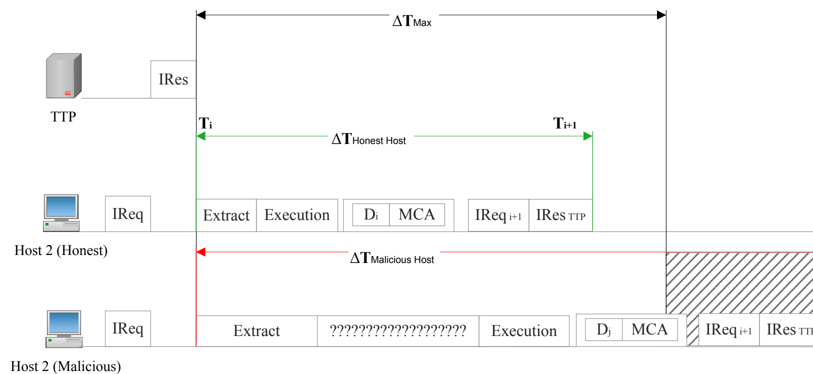

Fig. 11. Ejecución honesta vs ejecución maliciosa

que este host en particular ha ejecutado su variante asignada. En caso contrario, asumimos que el host ha ejecutado cualquier otra variante o ha ejecutado una versión alterada de la suya propia, y por lo tanto lo etiquetamos como sospecho.

3.5 Ejemplo de ataque por confabulación basado en code passing

Tal y como hemos contado, en un ataque basado en Code Passing, un host envía el código del agente a otro host que no es el siguiente destinatario en la ruta del agente, sino que es cualquier otro dentro del itinerario; gracias a esto, este host puede disponer del código del agente mucho antes de recibirlo a través de la ruta legal, y por tanto puede aprovechar ese tiempo para analizarlo en busca de posibles vulnerabilidades. Posteriormente, cuando el segundo host involucrado en el ataque reciba el agente a través de la ruta legal, intentará obtener una ejecución favorable. Para intentar evitar este tipo de ataques, usaremos nuestro Multi-Code Agent.

Supongamos que el itinerario de nuestro MCA esta compuesto por veinte hosts, y que dos de estos hosts tienen intención de realizar un ataque basado en Code Passing. En este caso, el host origen construirá un MCA compuesto por viente variantes del agente orginal (una por cada host del itinerario). La Figura 12 nos muestra una visión general del conjunto de acciones llevadas a cabo por los hosts maliciosos en su intento de realizar un ataque por Code Passing. El Host-1, recibe el MCA del host origen y pide las extracting instructions a la TTP. La TTP autentica al Host-1, verifica que puede enviar las extracting instructions en ese momento, y finalmente las envía. Una vez que el Host-1 ha recibido las *extracting Instructions*, envía toda la información necesaria al Host-13 (el MCA, las *extracting instructions* de la variante 1, o cualquier información que puede ser utilizada para atacar el agente. Después de esto, el Host-1 continua con la ejecución normal de su variante, obtiene los resultados de ejecución y envía el MCA al siguiente host (Host-2), quien extrae y ejecuta la variante B. De esta forma, el MCA irá siguiendo su ruta normal segun lo dispuesto en el itinerario

marcado por el host orígen, hasta llegar de nuevo al Host-13 para ejecutar la variante M. Mientras que se produce esta migración (vease Figure 12.a), el Host-13 aprovecha este tiempo para analizar la información privilegiada recibida a través de la ruta de confabulación (see Figure 12.b). Con esta información privilegiada (el MCA, las *extracting instructions* del Host-1, la variante A) el Host-13 puede intentar entender como funciona la variante M para alterarla antes de ejecutarla, o puede intentar usar la variante A.

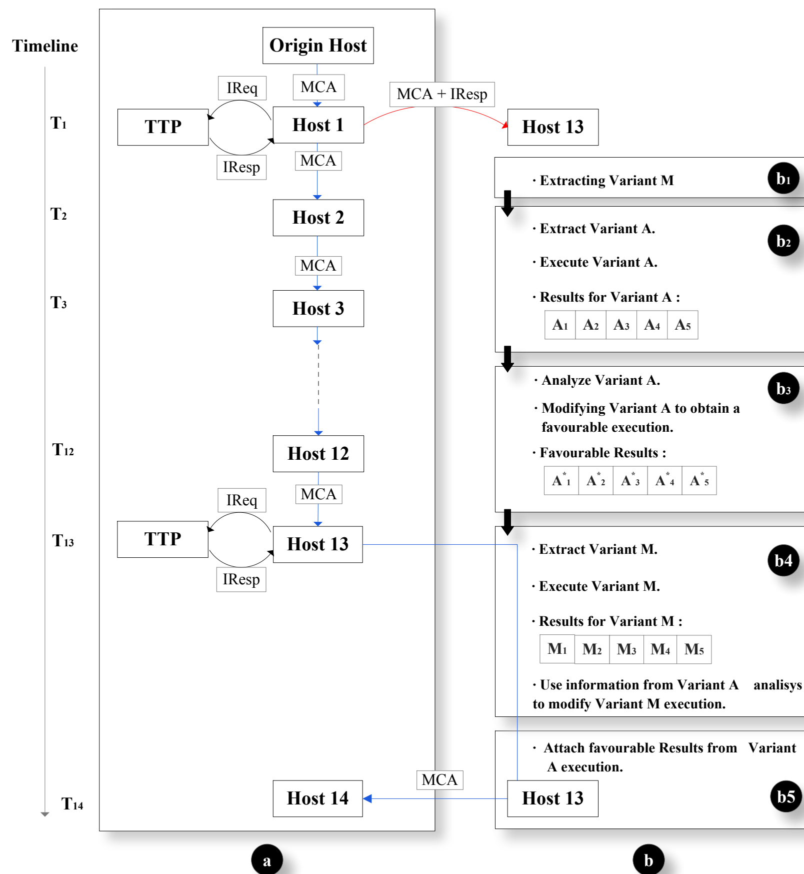

Fig. 12. A collusion attack based on code passing

El primer ataque que puede lanzar el Host-13, consiste en intentar averiguar que instrucciones del código del MCA pertenecen única y exclusivamente a la variante M (vease la Figura 12.b1). Este ataque es imposible que de resultado sin haver obtenido las correspondientes *extracting Instructions* previa petición a la

TTP. El Host-13 no dispone de suficiente información como para determinar que instrucciones pertenecen únicamente a su variante, que instrucciones pertenecen a otras variantes, y que instrucciones pertenecen a todas o a varias variantes.

El segundo ataque que puede llevar a cabo el Host-13 consiste en analizar la variante A para inferir cuales son las intenciones del agente, y intentar entender como se construye la Identity Mark. Con esto, intentaría generar un estructura de datos válida a partir de la información obtenida de la ejecución de la variante A. Como el Host-1 le ha facilitado las *extracting Instructions*, el Host-13 puede extraer la variante A, ejecutarla, y obtener el vector con los resultados de ejecución $[A_1 A_2 A_3 A_4 A_5]$ de forma rápida y sencilla, pero no obtendrá la información deseada puesto que este vector en particular no se correponde con la identity mark del Host-13 (ya que los valores dependen de la variante que se ha ejecutado). Aun así, el Host-13 puede intentar analizar estos resultados de ejecución para entender como se construye la Identity Mark (vease Figura 12.b2). Sin embargo, el análisis de una versión particular de una Identity Mark no le permitirá desentrañar la estructura de la Identity Mark. Tan solo le permitiran obtener los valores que compenen esta Identity Mark, pero sin saber a que hace referencia cada uno de ellos ni a que secuencia de ordenación están sujetos. Es decir, algunos de estos valores se corresponden con resultados de ejecución propiamente dichos (R_i), y el resto se corresponden con valores intermedios (V_i), pero de echo el Host-13 no sabe que valores se corresponden a uno a o a otro grupo ni en que orden. tal y como se muestra en la Figura 13.a, el Host-13 no puede extrapolar la relación entre los Host 13 A_i y los $V\,_i$ o los R_i. Por último, el Host-13 puede intentar analizar el código de la variante A, y modificarlo para obtener una ejecución favorable. Gracias a esta ejecución deshonesta, se generan unos resultados de ejecución alterados $[A_1^* A_2^* A_3^* A_4^* A_5^*]$ (vease la Figura 12.b3) que pueden ser comparados posteriormente con los resultados obtenidos honestamente usando la variante M. Para ello, cuando el Host-13 recibe el MCA desde el Host-12 (a través de la ruta legal), este finalmente puede pedir las *extracting Instructions* a la TTP y extraer la variante M. Ahora, el Host-13 tiene un tiempo limitado para para ejecutar la variante M, y por lo tanto no puede invertir demasiado tiempo en en realizar un análisis profundo del código. Es por esto, que la única cosa que puede hacer es ejecutar la variante M honestamente, obtener $[M_1 M_2 M_3 M_4 M_5]$, y usar la información extraída previamente para modificar estos valores convenientemente y obtener una Identity Mark válida (Figura 12.b4). Esto sería posible en el supuesto que tanto la variante A como la variante B, construyeran los valores de la Identity Mark de la misma forma y siguiendo el mismo orden, pero cada agente calcula estos valores de forma distinta y en un orden distinto. como consecuencia, el Host-13 no puede determinar la estructura de su Identity Mark en base a $[A_1^* A_2^* A_3^* A_4^* A_5^*]$ y $[M_1 M_2 M_3 M_4 M_5]$ (vease la Figura 13.b)

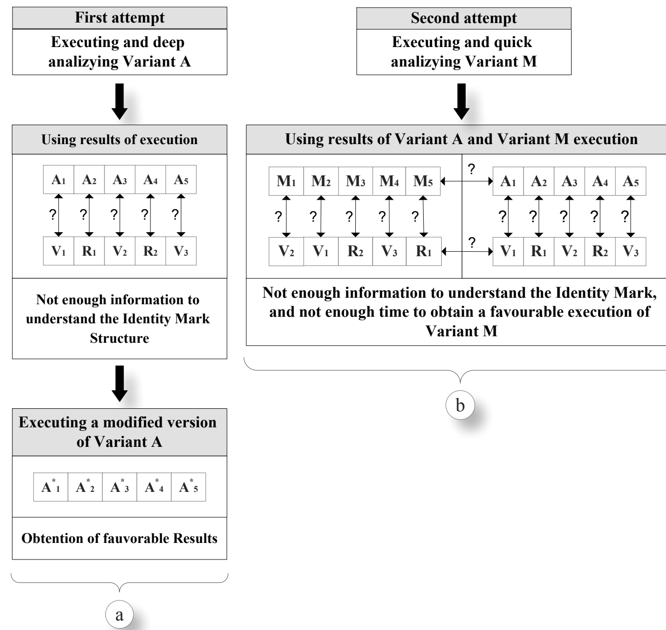

Fig. 13. Understanding the Identity Mark

4 Propuestas de ofuscación de código

Existen varias técnicas de ofuscación de código que podemos aplicar a un agente para dotarlo de mayor robustez ante un ataque. La ofuscación de código transforma un programa en otro semánticamente equivalente, pero que resulta más dificil de entender mediante ingeniería inversa. El problema que nos encontramos con las técnicas habituales de ofuscación, es que no las podemos aplicar directamente sobre el código del MCA porque está compuesto por varios códigos. Además tampoco las podemos aplicar sobre cada variante, porque entonces resultaría muy complicado construir el MCA. Por esto debemos aplicar un tipo de ofuscaciones específicas que no afecten al funcionamiento y a la codificación del MCA. En esta Master Thesis, se proponen tres posibles ofuscaciones que dotaran al MCA de mayor robustez ante ataques. En primer lugar veremos una ofuscación basada en insertar *Dummy Instructions*, que no són más que instrucciones basura que no se van a ejecutar pero que se insertan en zonas de código dónde tendría sentido usarlas. En segundo lugar propondremos una ofuscación consistente en desordenar los conjuntos de instrucciones dentro de cada sección de código entrelazado. Y finalmente, propondremos una ofuscación muy sutil consistente en hacer que las implementaciones del código de los métodos para variantes distintas, no difieran en su totalidad. Con esto conseguiremos que au-

mente el código compartido por las variantes y por tanto podremos aumentar el factor de compresión en la fase de Compressing.

4.1 Inserción de Dummy Instructions

Esta técnica de ofuscación de código tiene una particularidad especial, en lugar de ofuscar el código fuente, la ofuscación se realiza directamente sobre el código compilado (en este caso el Java Bytecode). Aunque la Máquina Virtual de Java verifica el código antes de ejecutarlo, nosotros no tenemos que preocuparnos por esto ya que cuando se extraigan los agentes se eliminaran estas instrucciones basura. Las *Dummy Instructions* son por lo tanto, instrucciones Bytecode válidas que se añaden directamente al Bytecode del MCA con el único objetivo de dificultar la comprensión del código. Retomando el ejemplo del MCA formado por dos agentes, vamos a analizar la información que podría extraer un atacante si no se realiza algún tipo de ofuscación. La Figura 14.a nos muestra la sección de código entrelazado del MCA final. Esta sección de código entrelazado contiene las implementaciones del method1() (vease la Figura 3) tanto de la variante A como de la variante B. Las celdas resaltadas en negro se corresponden con las instrucciones específicas de la variante A, las celdas resaltadas en gris se corresponden con las instrucciones específicas de la variante B, y las celdas blancas contienen instrucciones compartidas por ambas variantes. La instrucción en la posición 0 ($bipush10$) carga el valor entero 10 en la pila de operandos, y las instrucciones en la posición 2 y 3 almacenan este valor en la variable local 1 y 3 respectivamente. Hasta aquí, el código se de las dos variantes se puede distinguir con facilidad ya que ambas estan cargando el mismo valor (10) en distintas variables. Hasta la posición 10 ($iadd$), un atacante con conocimientos de Java Bytecode podría separar las instrucciones de cada variante, si nos fijamos, el agente A ha guardado el valor en la variable local cuyo índice es 1 ($istore_1$), mientras que el agente B ha guardado ese mismo valor en la variable local cuyo índice es 3 ($istore_3$). Posteriormente el agente A usa la instrucción $iload_1$ (almacenar en la pila el valor contenido en la variable local 1), y el agente B usa la instrucción $iload_3$. Es decir, cada agente almacenará y extraerá valores de la misma variable local. Para evitar que el atacante puede llegar a deducir esta información, podemos en primer lugar incluir una *Dummy Instrcuction* que cargue un valor arbitrario en la pila de operandos, por ejemplo bipush 20 (vease la Figura 14.b). Al poner esta instrucción inmediatamente después del $bipush10$, el atacante no sabrà a que valor hace referenecia la instrucción $istore_1$ y la instrucción $istore_2$. Si realizamos varias inserciones de este tipo a lo largo del código, el atacante necesitará invertir mucho tiempo para obtener las múltiples posibles combinaciones válidas, y de todos modos no sabría distinguir cual es la buena. Otra solución a este problema, consiste en añadir almacenamientos y extracciones de las variables locales (vease Figura 14.c). Si por ejemplo añadimos un $istore_2$ en la posición 3, y un $iload_2$ en la poición 8, el atacante nos sabe cual de estas instrucciones esta usando cada agente. Además si aplicamos inserciones de instrucciones $bipushbyte$ conjuntamente con inserciones de instruccionec $istore_i$

y *iload_i*, podemos conseguir ofuscaciones más elaboradas que harán que un atacante no pueda ni tan siquiera entender fracciones pequeñas del código.

Debemos tener en cuenta que el código mostrado en el ejemplo (vease Figura 14) corresponde a una implementación muy sencilla. Probablemente, si usaramos una agente real, el código de por sí ya sería considerablemente más complejo. Sin embargo, y teniendo en cuenta que nos podemos encontrar cualquier tipo de agente, hemos optado por considerar que el código resultante es sencillo de entender par ponernos en el peor de los casos. La ventaja de estas ofuscaciones al nivel del Bytecode, es que podemos ofuscar cualquier tipo de código de forma muy local. El inconveniente principal de esta técnica, es que resulta muy tedioso tener que ir insertando instrucciones una por una sobretodo cuando se traté de códigos de longitud considerable. Por todo esto, recomendamos usar esta técnica solo para corregir pequeñas secciones de código que pueden ser entendidas con cierta facilidad. Para dotar a todo el conjunto de código entrelazado de mayor opacidad, recomendamos usar la técnica descrita en 4.4.

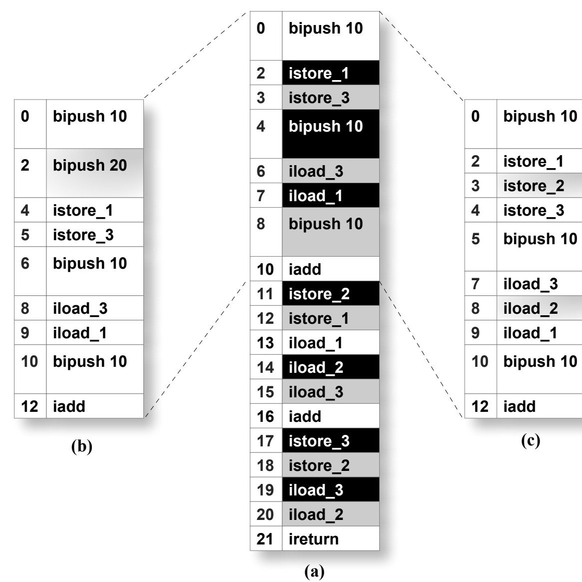

Fig. 14. Ofuscación por inserción de Dummy Instructions

4.2 Alteración del orden de las instrucciones

En la fase de Mixing, realizamos un entrelazado con las instrucciones específicas de cada variante, y de esta forma se generan conjuntos de instrucciones que pertenecen a distintas implementaciones, pero que a su vez hacen referencia al

mismo segmento de código en cada una de variantes. Dentro de estos conjuntos, las instrucciones pertenecientes a cada variante se pueden ordenar de forma distinta para cada segmento. Gracias a esta constante alteración del orden de las instrucciones, si se consigue descifrar el orden de un conjunto de instrucciones, el resto no se ve comprometido. La Figura 15.a nos muestra varios segmentos de código entrelazado dónde no se ha alterado el orden de aparición de las variantes. La Figura 15.b muestra varios segmentos de código entrelazado dónde se ha alterado el orden de aparición de las instrucciones.

202	Variant 1
203	Variant 2
204	Variant 3
205	Variant 4
206	Variant 5
207	Variant 1
208	Variant 2
209	Variant 3
210	Variant 4
211	Variant 5
212	Variant 1
213	Variant 2
214	Variant 3
215	Variant 4
216	Variant 5
217	Variant 1
218	Variant 2
219	Variant 3
220	Variant 4
221	Variant 5

(a)

202	Variant 1
203	Variant 2
204	Variant 3
205	Variant 4
206	Variant 5
207	Variant 3
208	Variant 2
209	Variant 5
210	Variant 4
211	Variant 1
212	Variant 5
213	Variant 1
214	Variant 3
215	Variant 4
216	Variant 2
217	Variant 1
218	Variant 3
219	Variant 2
220	Variant 4
221	Variant 5

(b)

Fig. 15. Ofuscación por alteración del orden de las instrucciones

4.3 Compartición de variables entre variantes

Nuestra propuesta se basa en generar múltiples variables, y usarlas de modo distinto en cada una de las variantes. De este modo, las instrucciones Bytecode que incluyen índices de variables locales en cada segmento de código entrelazado, son distintas para cada variante. No obstante, no es necesario que las variables usadas por cada variante en cada momento sean totalmente distintas. Al contrario de lo que puede parecer lógico, si las variantes comparten ciertas variables, conseguimos que el Bytecode resultante no sea totalmente disjunto y por lo tanto es más difícil para un atacante distinguir las instrucciones correpondientes a cada variante.

4.4 Precarga de variables

Retomando el problema planteado en 4.1, vemos que puede ocurrir que el código sea fácil de entender (vease la Figura 14.a). Esto es debido a que en el código fuente del agente, las variables se usan inmediatamente después de haber almacenado un valor en ellas. Por lo tanto, en el Bytecode se pueden distinguir las instrucciones de cada variante simplemente identificando que variable hacen referencia. Evidentemente, esta distinción no se puede realizar a nivel global, pero si que permite distinguir pequeños conjuntos de instrucciones pertenecientes a distintas variantes. Para conseguir eliminar del todo esta posibilidad, vamos a solucionar el problema de raíz haciendo una precarga de las variables en el código fuente. Recordemos que nuestro esquema de generación de las variantes se basa en la adición de múltiples variables al código original. Estas variables serán usadas en orden distinto por cada agente, de forma que el código resultante también será distinto. Sabiendo esto, asignaremos primero valores a varias variables y luego haremos las operaciones que las involucren. El resultado será que tendremos una amalgama de instrucciones $istore_i$ consecutivas, que en algunos casos haran referenacia al mismo índice, pero que pertenecerán a variantes distintas. Seguidamente tendremos instrucciones de todo tipo junto con las correspondientes $iload_i$. Con esta agrupación de instrucciones el atacante será incapaza de trazar el rastro de una variante identificando los índices de los $istore_i$ y los $iload_i$, y además podremos comprimir aquellas instrucciones que hagan referencia al mismo índice. Esta técnica esta indicada para ofuscar agentes cuyo código sea extremadamente sencillo, de longitud considerable. Su inconveniente, es que dependiendo de como este programado el código fuente, puede aumentar de forma considerable el tamaño del MCA. Sin embargo, combinando esta técnica con la presentada en 4.3 podemos generar muchas instrucciones compartidas por varios agentes, dándo lugar a instrucciones repetidas que pueden ser comprimidas en un única instrucción. La ventaja de usar estas técnicas combinadas, es que las *extracting instructions* estan diseñadas de tal modo que prácticamente no aumenta ni su longitud ni su complejidad.

5 Conclusiones

Este paper introduce un nuevo mecanismo para evitar ataques por confabulación basados en Code Passing. El mecanismo se basa en construir un Multi-Code Agent que contiene diferentes variantes del código de un agente original. Cada variante se ejecuta en un host distinto, y gracias a esto, cualquier información compartida por los hosts (en una confabulación) no sirve para obtener ninguna ventaja. La introducción de una Trusted Third Party, que se encarga de distribuir las *extracting Instructions* y tomar referencias temporales, nos permite detectar y evitar ataques por Code Passing. Aunque el Multi-Code Agent contiene diferentes variantes del código de un mismo agente, la longitud de este es prácticamente igual a la de un único agente. Por tanto, evitamos malgastar ancho de banda, que es un inconveniente típico de las propuestas basadas en el uso de varios agentes. Con la inclusión de una Identity Mark en los resultados

de ejecución del agente, podemos verificar que cada host ha ejecutado su correspondiente variante. Finalmente, el uso de la Trusted Third Party para obtener referencias temporales de confianza, nos permite limitar el tiempo de ejecución utilizado por los hosts.

Acknowledgements

This work was supported the Spanish Government through projects CONSOLIDER INGENIO 2010 CSD2007-00004 "ARES", TSI2007-65393-C02-02 "ITACA" and TSI2005-07293-C02-01 "SECONNET", and by the Government of Catalonia under grant 2005 SGR 01015 to consolidated research groups.

References

1. J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth. Cryptographic security for mobile code. In *IEEE Symposium on Security and Privacy*, 2001.
2. L. Benachenhou and S. Pierre. Protection of a mobile agent with a reference clone. *Computer Communications*, 29(2):268–278, 2006.
3. J. Borrell, S. Robles, J. Serra, and A. Riera. Securing the Itinerary of Mobile Agents through a Non-Repudiation Protocol. In *IEEE International Carnahan Conference on Security Technology*, 1999.
4. C. Cachin, J. Camenisch, J. Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In *27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*. Springer-Verlag, 2000.
5. D. Chess. Security considerations in agent-based systems. In *First IEEE Conference on Emerging Technologies and Applications in Communications (etaCOM)*, 1996.
6. C. Unger F. Kaderali D. Westhoff, M. Schneider. Methods for Protecting a Mobile Agent's Route. In *ISW'99*, volume 1729 of *LNCS*. Springer-Verlag, 1999.
7. O. Esparza, J.L. Muñoz, M. Soriano, and J. Forné. Punishing Malicious Hosts with the Cryptographic Traces Approach. *New Generation Computing*, 24(4):351–376, 2006.
8. W.M. Farmer, J.D. Guttman, and V. Swarup. Security for mobile agents: issues and requirements. In *19th National Information Systems Security Conference*, 1996.
9. S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.
10. F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.
11. F. Hohl. A Framework to Protect Malicious Hosts Attacks by Using Reference States. In *International Conference on Distributed Computing Systems (ICDCS)*, 2000.
12. W. Jansen. Countermeasures for Mobile Agent Security. *Computer Communications, Special Issue on Advanced Security Techniques for Network Protection*, 2000.
13. W. Jansen and T. Karygiannis. Mobile Agent Security. Special publication 800-19, National Institute of Standards and Technology (NIST), 1999.

14. H. Kim and L. Moreau. Trust Relationships in a Mobile Agent System. In *5th International Conference on Mobile Agents (MA'2001)*, volume 2240 of *LNCS*. Springer-Verlag, 2001.
15. D. Kinny. Reliable agent communication - a pragmatic perspective. *New Generation Comput.*, 19(2):139–156, 2001.
16. K.K. Leung and K.W. Ng. Detection Of Malicious Host Attacks by Tracing with Randomly Selected Hosts. In *International Conference on Embedded And Ubiquitous Computing*, volume 3207 of *LNCS*. Springer-Verlag, 2004.
17. A. Maña, J. Lopez, J.J. Ortega, E. Pimentel, and J.M. Troya. A framework for secure execution of software. *International Journal of Information Security*, 3(2):99–112, 2004.
18. Y. Minsky, R. van Renesse, F. Schneider, and S.D. Stoller. Cryptographic Support for Fault-Tolerant Distributed Computing. In *Seventh ACM SIGOPS European Workshop*, 1996.
19. J. Mir and J. Borrell. Protecting Mobile Agent Itineraries. In *Mobile Agents for Telecommunication Applications (MATA 2003)*, volume 2881 of *LNCS*. Springer-Verlag, 2003.
20. G. Navarro, S. Robles, and J. Borrell. Role-Based Access Control for E-commerce Sea-of-Data Applications. In *Information Security Conference (ISC'02)*, volume 2433 of *LNCS*. Springer-Verlag, 2002.
21. S.K. Ng. *Protecting Mobile Agents Against Malicious Hosts*. PhD thesis, The Chinese University of Hong Kong, 2002.
22. R. Oppliger. Security issues related to mobile code and agent-based systems. *Computer Communications*, 22(12):1165–1170, 1999.
23. J. Ordille. When agents roam, who can you trust? Technical report, Computing Science Research Center, Bell Labs, 1996.
24. A. Ouardani, S. Pierre, and H. Boucheneb. A security protocol for mobile agents based upon the cooperation of sedentary agents. *J. Network and Computer Applications*, 30(3):1228–1243, 2007.
25. J. Riordan and B. Schneier. Environmental Key Generation Towards Clueless Agents. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.
26. V. Roth. Mutual protection of cooperating agents. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1906 of *LNCS*. Springer-Verlag, 1999.
27. T. Sander and C.F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.
28. G. Vigna. Protecting Mobile Agents through Tracing. In *Proceedings of the Third International Workshop on Mobile Object Systems*, 1997.
29. G. Vigna. Cryptographic traces for mobile agents. In *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.
30. U. G. Wilhelm, S. Staamann, and L. Buttyán. Introducing trusted third parties to the mobile agent paradigm. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*. Springer-Verlag, 1999.
31. B.S. Yee. A sanctuary for mobile agents. In *DARPA workshop on foundations for secure mobile code*, 1997.
32. C.M. Yu and K.W. Ng. A flexible tamper-detection protocol for mobile agents on open networks. In *International Conference of Information and Knowledge Engineering(IKE'02)*, 2002.