(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

PCT

(43) International Publication Date
11 November 2010 (11.11.2010)

(10) International Publication Number
WO 2010/127438 A1

(51) International Patent Classification:
G06F 21/22 (2006.01) *G06F 9/44* (2006.01)
G06F 5/00 (2006.01)

(21) International Application Number:
PCT/CA2010/000666

(22) International Filing Date:
6 May 2010 (06.05.2010)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/175,945 6 May 2009 (06.05.2009) US

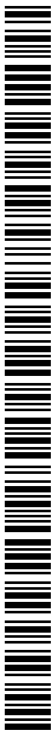
(71) Applicant *(for all designated States except US)*: IRDETO CANADA CORPORATION [CA/CA]; 84 Hines Road, Suite 300, Ottawa, Ontario K2K 3G3 (CA).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: **GU, Yuan, Xiang** [CA/CA]; 39 Insmill Crescent, Ottawa, Ontario, K2T 1G5 (CA). **MCRAE, Paul** [CA/CA]; 110 Columbia Avenue, Ottawa, Ontario K1V 1Z3 (CA). **NICOLESCU, Bogdan** [RO/CA]; 5761 Côte-Saint-Luc, Apt 306, Montreal, Quebec H3X 2E8 (CA). **LEVITSKY, Valery** [CA/CA]; 541 Kilbirnie Drive, Nepean, Ontario K2J 0E8 (CA). **ZHU, Xijian** [CA/CA]; 171 Gatespark Private, Kanata, Ontario K2T 1K9 (CA). **DONG, Hongrui** [CA/CA]; 32 Insmill Crescent, Ottawa, Ontario K2T 1G5 (CA). **MURDOCK, Daniel, Elie** [CA/CA]; 156B Valley Stream Drive, Ottawa, Ontario K2H 9C6 (CA).

(74) Agents: **MEASURES, Jeffrey M.** et al.; Borden Ladner Gervais LLP, World Exchange Plaza, 100 Queen Street, Suite 1100, Ottawa, Ontario K1P 1J9 (CA).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD,

[Continued on next page]



(54) Title: INTERLOCKED BINARY PROTECTION USING WHITEBOX CRYPTOGRAPHY

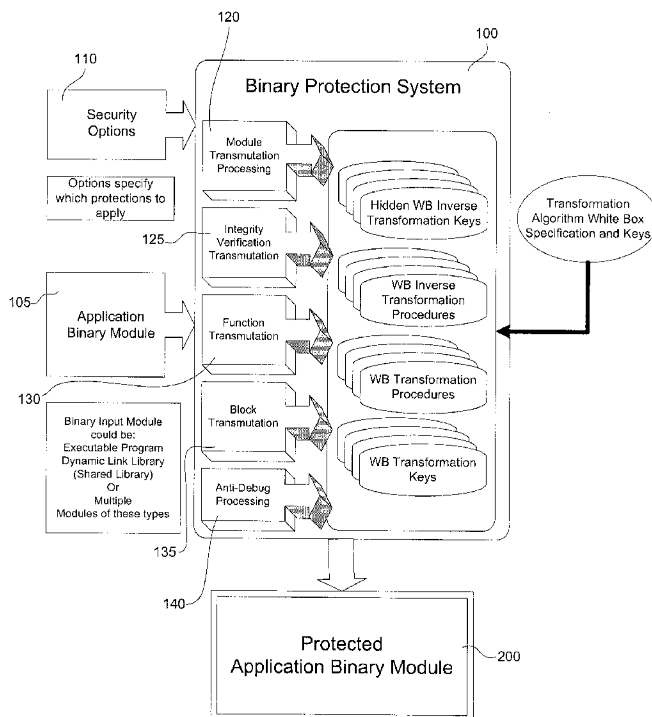


Figure 3

(57) Abstract: A system and method for transforming a software application comprising binary code and optionally associated data, from an original form to a more secure form. The method includes performing a combination of binary transmutations to the application, and interlocking the transmutations by generating and placing interdependencies between the transmutations, wherein a transmutation is an irreversible change to the application. Different types of the transmutations are applied at varied granularities of the application. The transmutations are applied to the application code and the implanted code as well. The result is a transformed software application which is semantically equivalent to the original software application but is resistant to static and/or dynamic attacks.

WO 2010/127438 A1

ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, RO, RS, RU, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*

INTERLOCKED BINARY PROTECTION USING WHITEBOX CRYPTOGRAPHY

FIELD OF THE INVENTION

[0001] The present invention relates generally to cryptography and computer security. More particularly, the present invention relates to a method for protecting binary applications from static and dynamic attacks. This application claims priority from US Provisional application US 61/175,945 filed May 6, 2009 which is hereby incorporated by reference in its entirety.

BACKGROUND OF THE INVENTION

[0002] In the information technology era, more and more applications will be deployed on billions of devices that are widely open for direct attacks. Stopping and preventing attacks have emerged among software designers as a major concern. Today's software is being cracked using increasingly more sophisticated tools, allowing average users to access sensitive information with little or no prior knowledge. Previous security models which assume an output is secure if a protected process operates in a black box (i.e., an attacker can not view the process which operates within the black box) is no longer effective. According to the attack's nature, security vulnerabilities are exploited by attackers, resulting in violation of one or more security properties.

[0003] Typically, debug-like activity represents the most common source of attack allowing hackers to understand the overall or detailed operations of an application. Once hackers obtain access to vital data and code, it is only a matter of time to figure out attack-points to focus their further attacks. Subsequently, attackers compromise system security either through the execution of clone programs, through the corruption of program binaries or simply by granting access to sensitive or copyrighted information (copying critical code partially or entirely).

[0004] Moreover, modern operating systems contain features that provide a good arsenal for other attacks such as memory lifting attacks. Attackers can take advantage of task management application program interfaces (APIs) offered by an operating system. For instance, the API that suspends processes characteristic to every multiprocessing operating system, allows attackers to access vital information while the process is frozen in memory.

[0005] While software security needs to address all these attacks, current security engines ignore many security factors and deliver security engines that concentrate exclusively on specific security concerns while completely ignoring other security threats.

[0006] For instance, anti-debug technology detects the presence of a debugger, but cannot protect applications against memory lifting attacks. Cryptography hides critical information, but does not, in and of itself, protect against debugging activity. Another instructive example is the integrity verification that protects applications against tampering attacks, but cannot prevent static or dynamic analysis attacks. As a general paradigm, each security technology addresses one or several attacks, but not the entire range of attacking domains.

[0007] It is, therefore, desirable to provide a security system that provides protection against a variety of attack domains. Furthermore, software obfuscation and transformation techniques have been proposed to transform the source code of an application. However, the present invention proposes enhanced security measures which include applying transformations at the binary level, including after the source code has been compiled.

SUMMARY OF THE INVENTION

[0008] It is an object of the present invention to obviate or mitigate at least one disadvantage of previous security systems.

[0009] One aspect of the invention provides a method of protecting a software application comprising binary code and optionally associated data, from an original form to a more secure form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said system comprising:

a. providing a build-time toolset to perform binary transmutation preparations to said application and transform the original execution of the said application to a secured execution by using the toolset; and
b. producing a protected application that is semantically equivalent to the original application, comprising interlocked transmutation executions, which also are interlocked with transmutation preparations, such that the binary protection is no longer separated from the protected application.

In this context the term “semantically equivalent” should be interpreted as “producing the same results or outputs”.

[0010] A transmutation is a change to the application which comprises more than a simple transformation, in the sense that a transformation typically comprises an operation for which there is an inverse operation to reverse the change made by the transformation, whereas a transmutation represents a more significant change for which there is no simple inverse operation. The transmutation comprises two phases:

- transmutation preparation, which applies a series of changes to the application code and implants new code intertwined with changed application code during build-time; and
- transmutation execution, which performs the prepared transmutation protection during execution of the protected application including implanted code.

[0011] Different types of the transmutations defined in (A) are applied at varied granularities of the application, with the granularities comprising the modules comprising the application, functions comprising a module, the basic blocks of instructions comprising the functions, and the individual machine instructions. The different types of transmutations include: module transmutation, function transmutation, block transmutation, instruction transmutation, integrity verification (IV) transmutation, and anti-debug transmutation. The transmutations incorporate one or more white box transformations.

[0012] The protection process is divided into two phases:

(A) a build-time phase, in which the software application is analyzed and transformed statically before execution; and

(B) a run-time phase, in which the transformed software application runs in memory.

[0013] One aspect of the invention provides a method of transforming a binary software application comprising binary application code from an original form to a secured form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

performing a combination of a plurality of binary transmutations to said binary software application during a build time phase by making a series of changes to said binary application code to produce changed binary application code, said changes including implanting new code intertwined with said changed binary application code during build-time; and

interlocking said transmutations by generating and placing interdependencies between the transmutations;

applying said combination of transmutations and interlocking to both the binary application code to be protected and the implanted code; and

producing a protected application that is semantically equivalent to the original application but which comprises said interlocked transmutations such that the binary protection is no longer separated from the protected application.

[0014] Another aspect of the invention provides a method of transforming a binary software application comprising binary application code from an original form to a secured form

that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

analyzing said binary application to determine at least one component of said application to which at least one binary transmutation can be applied, said component including component code;

performing a series of changes to said component code to produce changed component code, said changes including applying at least one WB transformation to said component code and implanting new code intertwined with said changes to said binary application code;

interlocking said changes by generating and placing interdependencies between said changes; and

applying said changes and interlocking to both the binary application code to be protected and the implanted code to produce a transmuted application that is semantically equivalent to the original application but which comprises said interlocked transformations such that the binary protection is no longer separated from the protected application.

[0015] Preferably, the secured execution of the said protected application comprises interlocked transmutation executions configured such that only a small portion of binary code is in clear form during any time of its execution.

[0016] At build time, the can system interact with the white-box transformation build-time facility to generate and assemble white-box transformation keys and operation code, and transform binary code and relevant important information from an input form to an output form by performing a white-box transformation operation with a white-box transformation build-time key.

[0017] At run-time, the system can interact with a white-box transformation run-time facility to transform binary code and relevant important information from an output form back to an input form by performing a white-box transformation inverse operation with a white-box transformation run-time key.

[0018] Other aspects and features of the present invention will become apparent to those ordinarily skilled in the art upon review of the following description of specific embodiments of the invention in conjunction with the accompanying Figures.

BRIEF DESCRIPTION OF THE FIGURES

[0019] Figure 1 illustrates white-box (WB) process of automatic WB code generation, according to an exemplary embodiment of the invention.

Figure 2 shows the automatic white-box key generation for transformation and reverse transformation processes, according to an exemplary embodiment of the invention.

Figure 3 depicts the proposed binary protection system with its foremost security components, automatically inserted into protected binary applications as interlocked security layers, according to an exemplary embodiment of the invention.

Figure 4 illustrates a schematic view of the protection process including the usage of a Transcoder as part of a pre-link protection process, according to an exemplary embodiment of the invention.

Figure 5 illustrates how source code transformations can be applied to both the code used by the application to be protected as well as the code comprising the binary protection techniques, according to an exemplary embodiment of the invention.

Figure 6 illustrates the structure of a protected binary application module, after buildtime processing, according to an exemplary embodiment of the invention.

Figure 7 illustrates the protection flow, where the input is a binary application, according to an exemplary embodiment of the invention.

Figure 8 depicts the protection flow of the Block Transmutation component, according to an exemplary embodiment of the invention.

Figure 9 illustrates the protection process with the Module Transmutation component, according to an exemplary embodiment of the invention.

Figure 10 illustrates the protection process with the Integrity Verification component that detects tampering attacks attempting to modify the behavior of the protected application, according to an exemplary embodiment of the invention.

Figure 11 is an overview of how the protected application executes securely at run-time, according to an exemplary embodiment of the invention.

5 Figure 12 illustrates the Module Transmutation (MT) component during run-time execution, according to an exemplary embodiment of the invention.

Figure 13 illustrates the run-time process of the Integrity Verification component, according to an exemplary embodiment of the invention.

10

Figure 14 depicts an example of automatic IV calls, by illustrating run-time function processing, both with and without IV, according to an exemplary embodiment of the invention.

Figure 15 illustrates the Function Transmutation (FT) component during run-time execution,
15 according to an exemplary embodiment of the invention.

Figure 16 illustrates the run-time process of the Block Transmutation (BT) , according to an exemplary embodiment of the invention.

20 Figure 17 illustrates an example of a protected application at different moments of its execution, according to an embodiment of the invention.

Figure 18 illustrates the build time process of the dynamic function loading according to an embodiment of the invention.

25

Figure 19 describes the run-time process of the Dynamic Function Loading, according to an embodiment of the invention.

Figure 20 illustrates an example of the first phase of the process of the Dynamic Function
30 Loading, according to an embodiment of the invention.

Figure 21 illustrates an example of the second phase of the process of the Dynamic Function Loading, according to an embodiment of the invention.

Figure 22 illustrates an example of the third phase of the process of the Dynamic Function Loading, according to an embodiment of the invention.

Figure 23 illustrates an example of the fourth phase of the process of the Dynamic Function Loading, according to an embodiment of the invention.

Figure 24 illustrates the final phase, in which, the calling application may request that the function be securely erased, according to an embodiment of the invention.

Figure 25 illustrates an example of the build time process for dynamic function loading with code transmutation and interlocking, according to an embodiment of the invention.

Figure 26 depicts an interlocked form of the BT and IV components, according to an embodiment of the invention.

Figure 27 illustrates an example of key splitting, in which, at protection time, the hidden decryption key is decomposed, according to an embodiment of the invention.

Figure 28 illustrates these different granularities or components of a program, which form successively nested layers, according to an embodiment of the invention.

Figure 29 illustrates the successively nested layers, and how the transmutations are applied to them, according to an exemplary embodiment of the invention.

DETAILED DESCRIPTION

[0020] Generally, the present invention provides a system and method for transforming a software application comprising binary code and optionally associated data, from an original form to a more secure form. The method includes performing a combination of binary transmutations to the application, and interlocking the transmutations by generating and placing interdependencies between the transmutations, wherein a transmutation is, in practice, an irreversible change to the application. Indeed, with respect to the level of difficulty involved, the combination of the interlocked transformations produces a transmutation to the application, which in practice, is irreversible, because the interlocking makes it not only difficult to reverse engineer the transformation applied, but simply applying the inverse of the transformation doesn't give you back the original code. To be clear, by irreversible, we do not necessarily mean that the entire process can not be reversed in a metaphysical sense, but rather that practically speaking, reversing the transmutation does not result in recovering the exact original program that was submitted for protection. Different types of the transmutations are applied at varied granularities or layers of the application. The transmutations are applied to the application code and the implanted code as well. The result is a transformed software application which is semantically equivalent to the original software application but is resistant to static and/or dynamic attacks. Note that we use the term "Implanting" to clarify that the process typically involves more than simple inserting of code. We use "implanting" to include making replacements and/or modifications to the original code in order to make the inserted new code workable within the changed application.

[0021] As stated, the invention applies a combination of transformations at varied granularities or layers, which are interlocked such that there is no simple reverse transformation. Figure 28 illustrates these different granularities, which form successively nested layers. Application 601 can include one or more modules 610, which encapsulate one or more functions 620, which encapsulate one or more blocks 630 which can encapsulate one or more instructions 640. Embodiments of the invention apply different transformations to each one of these successively nested layers, and also add interdependencies between them.

[0022] Embodiments of the invention join and interconnect various technologies within a post-link binary protection system that mitigates gaps in security architecture. In conjunction with white-box cryptography, embodiments of the present invention enable a defense strategy that comprises combinations of binary transmutations that are applied to the code to be protected. Embodiments of whitebox cryptography are disclosed in commonly owned United

States Patent No. 7,397,916 and US Patent Application No. 11/020,313 which are incorporated by reference herein in their entirety, as are commonly owned US Patent Nos. US 6,594,761; 6,779,114; and 6,842,862, and US Patent Application No. 11/039,817, which are incorporated herein by reference in their entirety.

[0023] These transmutations are described as follows:

- Anti-debug Transmutation (AD) checks if the protected application is being attached to a debugger.
- Block Transmutation (BT) provides binary protection from dynamic code analysis and attacks while protected application is running in memory. Its main role consists of splitting binary applications into white-box transformed blocks, then removing the transform upon request at run-time and subsequently destroying them after execution. Therefore, only the currently executing block of code is exposed.
- Function Transmutation (FT) acts as the block transmutation but applies the white box transform to functions which can contain many functional blocks, and removes the transform while invoking the function at run-time and destroys the function after return from the function.
- Integrity Verification Transmutation (IV) assures that the in-memory code segment of a protected application or any one of its components has not been tampered with and/or that the disk image of an application component has not been modified.
- Module Transmutation (MT) provides binary protection from static code analysis and attacks before the binary code is loaded into memory.
- Instruction Transmutation (IT) modifies individual instructions using a white box transformation such that the instruction must have the transform removed before execution. Further, these changes may even include changing the protected instruction itself (for example, inserting entry and exit handlers or by requiring an address manipulated by the instruction to be changed from its original value).

[0024] Embodiments allow the choosing of a set of security options, which determine how these transmutations are preferably automatically applied to binary application modules in an interlocked form. By interlock, we mean that we create dependencies within the application, such that two components which are interlocked each require the presence of the other in the final transformed application. For instance, integrity verification calls are inserted at each execution of a block or a function, such that block or function no longer works properly without a successful IV call, and that the IV call is only successful if the block or function has not been

tampered with. Furthermore, the security components themselves are protected by other security components (i.e. integrity verification protects the function transmutation engine and block transmutation protects integrity verification).

[0025] Except for anti-debug, the transmutations presented above can use white-box transformations to augment security. White-box transformations are distinguished by the fact that the key and transformation algorithm are protected even if the application is exposed to attackers. However, the key is never in its clear form since mathematical transformations are applied to the key that are known only by the transformation code. The white-box technology evoked in this specification is depicted by Figures 1 and 2.

[0026] Figure 1 illustrates the automatic code generation **10** for both transformation and reverse transformation processes, while Figure 2 shows the automatic white-box key generation for transformation **15** and reverse transformation **20** processes. The WB reverse transformation code **20** and hidden key **35** are generated from the same key and transformation algorithm **25**. In addition, each hidden WB key **35** relates to an inverse transformation **20** that reverses a transformation by a WB transformation code **15**. During the protection phase, users can use a variety of WB keys, allowing each security component to have its own set of white-box keys. For example, *integrity verification* and *module transmutation* components can use different keys. In addition, the *function transformation* component uses a multitude of keys to transform functions. To tune the security-performance trade-off, users have a wide variety of white-box transformation technologies, ranging from complex AES encryption algorithms to simple transformations such as XOR operations. This will be discussed in more detail in the key splitting section below.

Binary Protection System

[0027] Figure 3 depicts the proposed binary protection system **100** with its foremost security components, automatically inserted into protected binary applications **105** as interlocked security layers, according to an exemplary embodiment of the invention. Each security layer is implemented by executable code, whether a layer implements IV transmutation **125**, FT transmutation **130**, BT transmutation **135** or MT transmutation **120**. The security layers and the code they protect are both implemented by code that have predecessor/successor relationships allowing for the creation of interlocking relationships among layers. It should be appreciated that Figure 3 is schematic in nature. MT, FT, BT and instruction transmutations (IT) (not shown) are preferably arranged in successively nested layers (i.e. MT encapsulates

FT, which in turn, encapsulates BT, which in turn encapsulates IT, as shown in Figure 28) in a series of inter-layer transmutations, whereas IV 125 and ADB 140 can be applied on intra-layer basis (i.e, applied to each nested layer, or to a group of layers, or to the whole protected application).

[0028] Moreover, security layers execute in a sequence. Each layer, when it gains control, can be set to depend, for its proper execution, on values produced by a preceding layer. For example, MT loads executable code with specific errors that are corrected by FT or BT at execution time. The binary protection tool accepts as input a set of security options **110** that enable and provision the desired security components. Furthermore, users provide transformation keys that are coupled with the security components. In addition, the binary tool allows users to associate a set of transformation keys to a security component. For instance, the *Function Transmutation* component transforms functions with different keys.

[0029] An embodiment provides the ability to interlock with high-level source code protection. For instance, users can apply transformations in the source code in order to increase security. These high-level transformations are automatically applied by the transcoder. Details of the transcoder may be found in commonly owned US Patent Nos. US 6,594,761; 6,779,114; and 6,842,862, and US Patent Application No. 11/039,817, which are incorporated herein by reference in their entirety.

[0030] Figure 4 illustrates the schematic view of the protection process including the usage of a Transcoder **220** as part of a pre-link protection process, for embodiments in which the source code is also transformed. The *Binary Protection System* **100** box can be thought of as including the entire block diagram depicted in Figure 3. In addition, at the source level, users insert some security mechanisms, but the protections they provide are not complete until binary level explicit security engines are inserted as well. For example, in the source code, users employ some security APIs, but the resulting binary, after linkage, is made more secure after the corresponding security component is inserted at the binary level.

[0031] In Figure 4 the *Binary Protection Libraries* **230** represent the interface between users and security components. It is a static library providing a set of APIs to support security requests made by the program. Such APIs provide *integrity verification* and *anti-debug* functionality. Another important module in the binary protection process is the *Resident Security Module (RSM) **300*** that contains the five foremost security component engines. The RSM will be discussed in more detail below. During the post-link protection phase, the *Resident Security Module* **300** is packaged with the application binary module **200** (known as the payload). The

payload module is decomposed into white-box transformed blocks or functions, then compressed and transformed as a whole and packaged with the RSM into a protected application. The new protected application replaces the original executable program and shared libraries in the deployed application. In addition, the *Resident Security Module* contains the run-time engines of the main security components, the white-box inverse transformation code procedures as well as the transformed keys. The entire post-link protection process is automated. Except when making security options choices, users do not interact with the *Resident Security Modules*. Both *Binary Protection Libraries* and *Resident Security Module* are high-level code protected, and incorporate source code transformations as outlined in Figure 5. As shown in figure 5, the source code transformations can be applied to both the code used by the application to be protected as well as the code comprising the binary protection techniques. That is, both libraries linked in to the application to be protected and the code inserted into the application during the protection phase can have enhanced security by using source code transformations to make attacks difficult.

[0032] As stated above, the methods and systems described herein generally can be divided into build-time features/components and run-time features/components. Figure 6 illustrates the structure of a protected binary application module, after buildtime processing, according to an exemplary embodiment of the invention. Fundamentally, the structure of a protected binary application is composed of a Resident Security Module 300 and its transformed payload 350. The Resident Security Module implements static and dynamic protection. It contains security components that do not require the user interaction, such as the white-box reverse transformation routines, the automatic Integrity Verification and Anti-Debug engines, and the Dynamic Block Permutation Manager. In addition, the Resident Security Module is responsible for loading the payload into memory during the run-time phase.). It then stays resident (or portions of it do) so that it can provide services to the various binary transmutations. For example, upon entering an entry handler for function protection, the handler can call into the RSM to get the rest of the function decrypted, and so on.

Build Time Processing

[0033] Figure 6 can best be understood by discussing the process for building such a protected binary application module, namely the build-time process._This section describes the protection phase of each security component and how it is applied individually. The protection flow and the structure of protected binary applications are depicted and analyzed. In addition,

this section points out the interaction between security components. Generally, these security components are applied automatically, directly to the binary applications. Security engines of the security components reside in the *Resident Security Module* **300**. There is **typically** no user interaction demanded to install the security components. The binary application is massively transformed and is included as a payload **350** of the *Resident Security Module*. Thus the payload, for example, includes the transformed functions 410 (as well as entry and exit handlers (not shown) for each transformed function). According to the desired security technology, applications are modified at the binary level and transformed in several stages. When binary applications are protected with the Module Transmutation, the structure of the payload is not important; the payload is seen as a single entity.

[0034] However, the *Integrity Verification* and *Anti-Debug* components can have an additional mode that requires user interaction. In this case, functional engines of the security components reside in *Binary Protection Libraries* **230** that are statically linked into the protected binary application. As an important note, a security component can be applied in both modes. For instance, the *Integrity Verification* component can be applied at the link phase **103** if users apply the *Integrity Verification* APIs. Furthermore, the *Integrity Verification* can be applied again to the resulting binary as a part of post-link protection without any conflict.

Function Transmutation-Build Time

[0035] Figure 7 illustrates the protection flow, where the input is a binary application, according to an exemplary embodiment of the invention . The *Function Transmutation* is a post-link protection technology that prevents applications from dynamic attacks. The original binary application **105** is decomposed into its composite functions (e.g., Function 1, Function 2, Function 3....Function N), which are then transformed with WB algorithms by FT processing block **130.** The FT processing 130 is executed by a white-box transformation build-time facility 140, which generates and assembles white-box transformation keys and operation code for each function. Each transformed function 410 is then included as the payload of the *Resident Security Module*. The FT processing also installs *Entry* and *Exit* handlers **420** for each function. Each function is transformed with its own white-box key (and its own transformation code), and therefore, the resulting protected binary module contains a hidden WB reverse transformation key (transformation code) for each function. At this point, to avoid any confusion between the terms key and code, we clarify that code refers to the algorithm/software code used to perform

a transformation, which typically uses a unique key.

[0036] A benefit of the *Function Transmutation* component is to limit the number of functions that are in clear form at any given moment during execution. In addition, the *Resident Security Module* intercepts calls to transformed functions **410**, and co-operates with the *Entry* and *Exit* handlers **420** for each function. *Entry* handlers are responsible with reversing the WB transformation on the function and responsible for white-box key management, while *Exit* handlers destroy the functions in memory. Thus, calls to functions are performed via *Entry* handlers while returns from functions are done via *Exit* handlers. Apart from the operations specific to the *Function Transmutation* component (Function Transmutation and destruction), *Entry* and *Exit* handlers perform various operations: block permutation, integrity checks, test the presence of a debugger, etc. It should be noted that the figures are schematic in nature, and are included to illustrate how the system works, and are not meant to be all inclusive. Thus, Figure 6 does not show the Entry and exit handlers (mostly to avoid being too cluttered), whereas figure 7 only shows the portions of the RSM and payload relevant to the transformed functions themselves.

Block Transmutation – Build Time

[0037] Figure 8 depicts the protection flow of the *Block Transmutation* component **135**, according to an exemplary embodiment of the invention, which provides binary protection from dynamic code analysis and attacks while protected application is running in memory. *Block Transmutation* modifies the binary application by inserting and replacing binary instructions. Moreover, the binary application code is decomposed into white-box transformed blocks which need to be transformed back into an executable state in order to be executed, which occurs upon request at run-time. The BT processing block 135 decomposes the application binary module into its requisite functions, and then breaks each function into blocks. It then transforms the blocks, using a white-box transformation code. The BT has the biggest impact on the application binary structure. Functions that are protected with BT are significantly modified. For instance, some of the original instructions are replaced with new instructions, and some others are relocated within the function address space. Since new instructions are inserted, the size of each function protected with BT is larger. Hence, BT protected functions are relocated into a dedicated binary location. At run-time, functions protected with BT are comprised of a large code segment divided into WB transformed blocks **440**. Blocks are divided into groups, and

each group of blocks is transformed with a unique white-box key. The resulting binary application keeps its original behavior but is much more tamper resistant. The transformed white-box keys and reverse transformation routines are part of the protected binary module. In addition, the original application's control flow is disrupted and changed constantly during program execution thus preventing binary applications from automated attacks. The *Resident Security Module* contains a dynamic block permutation manager **155** that randomly modifies the blocks' physical memory address. The white-box transformed blocks are included in the payload mated with the *Resident Security Module*.

<u>Module Transmutation – Build Time</u>

[0038] Figure 9 illustrates the protection process with the *Module Transmutation* **120** component that protects binary applications from static analysis and attacks, according to an exemplary embodiment of the invention. In general, the protected application can be composed of several modules (executables and shared libraries), in which case this process is repeated for each module such that the protected application includes multiple transmuted modules. The original binary application **105** is transformed, compressed, and then included as the payload of the *Resident Security Module*. The resulting protected binary module replaces the original module in the deployed application. The right-hand box in Figure 9 reveals the structure of the application binary module **200** protected with the *Module Transmutation*. Apart from the functionality to load applications into memory, for static protection, the *Resident Security Module* **300** contains procedures to perform the reverse transformation to the payload **360** at execution time and includes integrity verification capabilities. Figure 9 only illustrates a portion of the payload 360, which forms part of the payload 350.

<u>Integrity Verification – Build Time</u>

[0039] Figure 10 illustrates the protection process with the *Integrity Verification* component that detects tampering attacks attempting to modify the behavior of the protected application, according to an exemplary embodiment of the invention. At build-time the input binary module is digitally signed, creating *voucher data* The voucher contains a collection of signatures and other related information about the protected application. The voucher data is transformed by IV process **125** using white box algorithms and the transformed voucher data is either embedded into the protected application or exported into a separate file. In general, the protected application can be composed of several modules (executables and shared libraries).

In this case, making the application tamper-resistant means that all modules are signed and the voucher data can be stored in a single entity for all modules or preferably, each module has its own voucher data stored independently. Allowing a voucher data for each module not only provides flexibility for the application to update individual modules without re-signing, but also increases the security by allowing each voucher to be transformed with a different key.

[0040] The *Integrity Verification Transmutation* can run in automatic and/or user-interaction modes. Both modes share behavior and principles, but they differ in the way the IV mechanisms are inserted into the protected binary application. In the automatic mode, an IV engine **540** resides in the *Resident Security Module* and IV components are inserted automatically into the protected binary application at strategic points. In the case of the user-interaction mode, the IV mechanism **550** resides in the *Binary Protection Libraries*, which are statically linked into the application, based on Calls to IV Library **555** which is inserted into the user's source code before the compilation phase.

[0041] Typically, at run-time the structure of protected applications consists of a large number of code segments that dynamically change their state depending on the different binary transmutations that have been applied. Indeed, the protected binary application structure comprises a combination of a plurality of binary transmutations including at least one inter-layer transmutation applied to successively nested layers of said binary application code, with the successively nested layers including the modules comprising the application, the functions comprising a module, the basic blocks of instructions comprising the functions, and the individual machine instructions. Most security-critical functions are protected with BT, and some others with FT. Despite the self-modifying code nature of protected applications, the IV transmutation preferably provides the ability to check integrity of a module, a function or a block. Therefore, the signing phase consists of associating a set of signatures to each code segment that changes its state during execution. They are signed prior and after being transformed. For instance, equation 1 illustrates the signature of block in its both transformed and untransformed states. Consequently, signature of a function protected with BT can be expressed by equation 2, which is a sum of block's signatures that compose the function. When functions are protected with FT, they are transformed as a single piece. Equation 3 represents the signature of a function protected with FT.

$$S_{Block} = \{S_{Transformed_Block}, S_{Untransformed_Block}\} \qquad (1)$$

$$S_{Function_BE} = \sum \{S_{Transformed_Block}, S_{Untransformed_Block}\} \quad (2)$$

$$S_{Function_FE} = \{S_{Transformed_Function}, S_{Untransformed_Function}\} \quad (3)$$

[0042] Before we discuss exemplary embodiments of the run-time processing we refer to Figure 29, which is a different schematic illustrating the successively nested layers, and how the transmutations are applied to them, according to an exemplary embodiment of the invention. In this example, we are illustrating only one module, although it should be appreciated that additional modules may be involved. In this example, the module is divided into 3 functions. We will only discuss one exemplary function, namely Function 1, labeled as 770. This function comprises a plurality of blocks, each of which comprise a plurality of instructions. The FT processing block transmutes function 1 into a transformed form 775, which includes entry handler 773, and exit handler 778, and will no longer operate without interlocking code/data from the RSM and WB transformation engine 810. However, Function 1 comprises multiple blocks, each of which are also transmuted. For example, the BT processing block transmutes a block into a transformed form 780, which includes entry handler 783, and exit handler 785, which also will no longer operate without interlocking code/data from the RSM 790 and WB transformation engine 810. Similarly Block 780 includes one or more instructions, which are transformed into a transformed form 790 which also will no longer operate without interlocking code/data from the RSM 790 and WB transformation engine 810. Further, the RSM 800 not only incorporated the successively nested interlayer transmutations discussed above, but also incorporates IV engine 820 and ADB engine 830 which add intra-layers of protection to one or more of said layers, and/or to the entire transmuted module.

[0043] Note that the above was described in a top-down order. That is to say, as Functions encapsulate blocks, we described FT prior to BT. However, it should be appreciated that in practical terms, embodiments first analyze the functions, and decomposes each function into blocks, transforms the blocks , effectively transforming the function. FT processing is then applied to the transformed function, as described above. At runtime, the process would be reversed, with the function transformation removed first, exposing the transformed blocks and the transformed blocks having their transformations removed as they are executed.

Run-Time Processing

[0044] Figure 11 is an overview of how the protected application executes securely at run-time. As is shown in Figure 6, the protected binary application is composed of a RSM **300** and its transformed payload 350. The structure of the payload is dependent on the binary transmutations applied at build time. When the protected binary application starts, the RSM takes control and prepares the payload to run. Effectively, the RSM replaces the OS loading process. The RSM contains information about the protected application and allocates memory space where the transformed payload will be located when the reverse transformation is applied. Prior to dealing with the transforms on the payload, the RSM performs three operations: 1) initializes the security engines, 2) runs the *Anti-Debug* component and 3) checks the integrity of the payload.

[0045] After the payload has been set up in memory, the RSM prepares the run-time security support and then transfers control to the payload, which is composed of transformed functions and transformed blocks. Additionally, the application contains integrity information related to blocks, functions and the entire application module. Functions and blocks are dynamically transformed into an executable state upon request and destroyed after execution. As a result, only a fraction of code will be in clean form at run-time. During the initialization phase, the RSM manages *Entry* and *Exit* handlers, which are responsible for function and block transformation and destruction. In addition, these handlers perform other security activities, such as integrity checks, code relocation, code deploying and the detection of a debugger attached to the protected application. Integrity verification allows for automatically checking the integrity of a block, a function or the entire application.

[0046] Code relocation capabilities mean that after each function invocation, the function changes dynamically in structure. This is possible by permuting the function blocks' physical memory address. Dynamic code relocation represents an excellent defense against automated and dynamic attacks. As the *Entry* and *Exit* handlers are key elements in offering dynamic protection, they are also preferably protected themselves. Each handler can check its own integrity and re-structure itself periodically. In addition, the handler checks if any debugger is attached to the protected application.

Module Transmutation – Run-time Process

[0047] Figure 12 illustrates the Module Transmutation (MT) component during run-time execution. When the protected application is loaded into memory, the RSM takes control and performs some security operations. Firstly, the RSM verifies that the protected application is not under debugger control and checks integrity for itself and the payload then transforms the payload into a state ready for execution. The MT component is primarily intended for static protection. It prevents attackers from analyzing the payload on the disk. After the payload is deployed, other security components embedded into the payload and in the RSM, take control and protect applications from dynamic analysis and attacks. In addition, the payload contains run-time data related to the binary protected application.

Integrity Verification Transmutation – Run-time Process

[0048] Figure 13 illustrates the run-time process of the Integrity Verification component. When the protected application is deployed into memory, its structure is comprised of code segments that are dynamically transformed into an executable state upon request and destroyed again after execution. The IV Transmutation (IV) ensures that the in-memory code segment of a protected application or any of its components have not been tampered with and that the disk image of the application has not been externally modified. The basic principle relies on comparison of integrity verification data generated at protection/build time with the run-time image of the protected binary application. Every time a block or function changes its transformed state, the run-time signatures are updated appropriately. Therefore, it is possible to check the integrity of self-modifying code applications, such as binary applications that are composed of segments that dynamically change their transformation states.

[0049] In automatic mode, the IV can verify integrity of a module, a function or a block. According to the desired security level, calls to IV are inserted at strategic points in various forms. For instance, when functions are protected with the Block Transmutation (BT), prior to each block execution, the IV checks the block's integrity or the integrity of all blocks that compose a function. Checking the integrity of a function means checking the integrity of all blocks that comprise that function, while the integrity of a module is comprised of multiple signature values that belong to BT/FT protected functions as well as the rest of the application code. Figure 14 depicts another example of automatic IV calls, by illustrating run-time function processing, both with and without IV. Using a dual signature scheme, functions can be verified prior to execution, which represents an excellent defense strategy. Figure 14.a) represents a

basic call to an FT protected function, while 14.b) illustrates a call to an FT protected function containing integrity checks of the both the caller and called functions.

[0050] In user mode, the IV provides a set of APIs that check the integrity of a module or a function. In addition, users can control when IV checks occur by inserting them into the source code. The Integrity Verification functionality allows the user to check the integrity of a module in several ways. First, the entire unchanging set of bytes that comprise the application (i.e. the code and read-only data) can be checked whenever the corresponding IV API is invoked. This API allows the user to specify how much of the application is to be checked on each invocation for the purposes of tuning the performance. In addition to this overall module checking, the IV API allows the user to check if the function that is currently executing has been modified since build time. As described above, functions and blocks that are dynamically transformed and executed can be checked for integrity before and after execution.

Function Transmutation – Run-time Process

[0051] In general, during the application module execution, when a protected function is invoked, its entry function handler is first executed and interacts with the white-box transformation run-time facility to inverse-transform the white-box transformed function code by executing the white box transformation inverse operation using the white box transformation run-time key data and operation codes specific to the function; loading the inverse-transformed functions into execution memory; and transferring the execution control to the function and, upon exit from the function, invoking the function's exit handler. Optionally, RSM, for each function, clears or scrambles the function's memory footprint before returning. Additionally, intra layer protection can be invoked such that one or both of an IV or ADB transmutation is interlocked with the FT applied to a function being protected by applying the IV, ADB or both transmutations to the entry and exit function handlers implicitly.

[0052] Figure 15 illustrates the Function Transmutation (FT) component during run-time execution, according to an exemplary embodiment of the invention. Even after the RSM transforms the payload into an executable state, some functions can remain transformed. In addition, the RSM installs Entry and Exit handlers for each protected function. When a protected, transformed function is invoked, its Entry handler gains control and performs the appropriate inverse transformation to restore the function to an executable state using the necessary white box key. As stated earlier in this document, each function can be transformed using a unique key. Prior to performing the transformation operation, the Entry handler, if desired, performs various other operations (integrity checks, anti-debug,), as specified at protection time. After transforming the function into an executable state, the Entry handler transfers the control to the function for execution. The return from the function is intercepted by the function's Exit handler, whose main role is to destroy the function content and return control to the caller. As in the Entry handler, the Exit handler can perform a variety of security operations as specified at protection time.

[0053] The FT component allows users to define a dynamic counter, which specifies the frequency of functions' run-time destruction/transformation. By default, the counter is set to 1. Every function has its own dynamic protect counter. This counter allows the user to tune the protection for security vs. performance. If the counter is set to 5, for example, the function can be invoked 5 times before the Exit handler destroys the function content.

Block Transmutation – Run-time Process

[0054] Figure 16 illustrates the run-time process of the Block Transmutation (BT) component which is charged with protection against dynamic analysis and attack. The central principle relies on the decomposition of binary applications into discrete blocks. At build-time, the blocks are transformed using white box transformations. At run-time, blocks are dynamically transformed back into an executable state upon request. Preferably only a minimal number of blocks (and ideally only one block) will be in clean form at run-time, with remaining blocks still un-transformed. After execution, blocks are destroyed, BT and FT share the same basic principle: transformation-execution-destruction, but they are implemented differently. Functions protected with BE are modified at build time. For instance, calls to transformation routines are inserted within the protected function.

[0055] Dynamically transforming blocks instead of entire functions constitutes an additional layer of protection over and above that offered by the FT. For example, according to the function's input parameters, some blocks are not executed. Consequently, they are never transformed into an executable state and vulnerable to analysis, lifting, and tampering.

[0056] In addition, blocks that belong to the same function are constantly permuted in memory. Dynamically transforming blocks and permuting their position in memory is an excellent defense strategy against automated and dynamic attacks. Figure 17 illustrates an example of a protected application at different moments of its execution. The frequency of block permutation can be defined by users among with the input security options.

Dynamic Function Loading

[0057] In addition to the technologies outlined above, the technology infrastructure provided by this system can also enable the dynamic loading of protected functions. This dynamic loading functionality allows the user to protect the function using a White Box transformation, load the function when needed, invoke the function, then securely erase the function from memory. In addition to allowing for easy function updates without changing the base application, this system minimizes the time the protected function is in the clear in memory.

Dynamic Function Loading - Build Time

[0058] Figure 18 illustrates the build time process of the dynamic function loading. The

user supplies the functions that are to be protected. These are processed through the Dynamic Function Loader processing engine in conjunction with the user-supplied White Box transformation key and algorithm specification. The process produces a protected data module that contains the transformed functions and the hidden white box inverse transformation key. This data module can then be supplied to the application in a variety of ways (download, etc.). This allows the user to easily change the nature or protection level of the functions without having to rebuild, retest, and redeploy the original application. It also allows the user to place function code (in the form of this data module) into areas on the target system that might not normally allow for code binaries such as executables and dynamically loaded or shared libraries.

Dynamic Function Loading - Run-time Process

[0059] Figure 19 describes the run-time process of the Dynamic Function Loading. At runtime, the application makes a call into the Dynamic Function Loading library specifying which protected data module is to be loaded and which function is to be invoked. The library then finds the function in protected data module and transforms it into an executable form into memory. The library then invokes the function on behalf of the application and returns the results. After invoking the function, the application can then ask that the function be securely erased from memory.

[0060] The run-time process of the dynamic function loading is divided into several phases. Figure 20 illustrates an example of the first phase of the process. In this phase, the calling application specifies which protected data module contains the desired function. The Dynamic Function Loading Library then identifies the function in the library.

[0061] Figure 21 illustrates an example of the second phase of the process. In this phase, the In this phase, the Dynamic Function Loading Library obtains the hidden White Box transformation key and prepares to apply the inverse transformation the desired function. Figure 22 illustrates an example of the third phase of the process. In this phase, the data containing the code for the desired function is transformed into an executable state, placed into memory and prepared for invocation.

[0062] Figure 23 illustrates an example of the fourth phase of the process. In this phase, the Dynamic Function Loading Library invokes the function on behalf of the calling application. The application has supplied the parameters to the function, which are passed on by the library. The return code from the function is then sent to the caller after the function completes. Figure

24 illustrates the final phase, in which, the calling application may request that the function be securely erased. The library then erases the function from memory. If the function needs to be invoked again, it must again be transformed into an executable state and loaded by the library.

[0063] Figure 25 illustrates an example of the build time process for dynamic function loading with code transmutation and interlocking. Dynamic Function Loading provides an infrastructure for greater security techniques to be applied. For example, any of the function protection techniques outlined in previous sections can be used:

- Block Transmutation
- Function-based Integrity Verification
- Block-based Integrity Verification
- Antidebug

[0064] Greater security can also be obtained by combining Dynamic Function Loading with source code transformations. In the example illustrated in Figure 25, the user supplies an application containing 3 functions that need to be protected. By applying source code transformations via the Transcoder, the system can create an application containing transformed stubs for the functions that are strongly interlocked with the transformed functions. These functions can then be protected using the Dynamic Function Loading mechanism. Using code transformations in this manner will make it very difficult for an attacker to replace the protected functions while the Dynamic Function Loading technique makes it difficult to lift the protected functions from memory. In addition, this technique allows the user to replace the protected functions easily, perhaps to respond to an attack on the initial implementation.

Interlocking Security Technologies

[0065] Interlocking aims at raising the security standards by creating strong correlation between various security technologies. It focuses on critical code and data. Key components of a certain security engine depend on the values produced by other security component. Such an example of interlocking security engines is illustrated in Figure 14, where the integrity is checked prior the dynamic transformation of the verified function. Nevertheless, interlocking can be much more complex. For instance, IV and AD components provide Fail and Success callback routines that contain parts (executable code) of another security component.

[0066] For example, Figure 26 depicts an interlocked form of the BT and IV components. Part of the instructions that compose a block are relocated into the success callback of the IV engine. Therefore, they are executed depending on the success or failure of

the verification. In case of IV failure, the computation is not complete and the hacker attack will fail.

Key splitting

[0067] An important security paradigm in cryptography consists of protecting the decryption key. Using white box technology is a huge step forward in meeting this requirement. Mathematical transformations are applied to the decryption key that known only by the decryption routine, resulting in a decryption operation that does not expose the key even if the attacker is observing each instruction execute. The present invention introduces an additional security measure that consists of splitting the hidden decryption key into 2 parts. One internal part, embedded into protected binary application, and an external part that end-users must provide at the execution phase. Figure 2 illustrates the process of generating both the encryption **30** and hidden decryption key **35**. At protection time, the hidden decryption key is decomposed as depicted in Figure 27. At run-time, the RSM is in charge of the key reconstitution. The technique extends to any white box transformation. The key splitting feature applies to all security components that use WB transformations: MT, IV, FT and BT.

[0068] For example, the manufacturer of a device can place part of the key in a secure store on the device. When a protected application is invoked, this part of the key can be retrieved by executing code on the device that retrieves the key from the secure store. The complete White Box key can now be reconstituted and used to decrypt the payload of the protected application, used to decrypt voucher data, or used for function and/or block transformations. An attacker getting the key fragment from the secure store cannot do anything with it, neither can an attacker that manages to extract the key fragment from the application. Additional use of interlock technology can make it very difficult to perform any decryption operations outside of the context of the protected application running on the device with the secure store. Of course, transformations on the key make it difficult to use outside of the white box context, i.e., an attacker cannot use a transformed key with their own encryption routines.

[0069] In the preceding description, for purposes of explanation, numerous details are set forth in order to provide a thorough understanding of the embodiments of the invention. However, it will be apparent to one skilled in the art that these specific details are not required in order to practice the invention.

[0070] Embodiments of the invention can be represented as a software product stored

in a machine-readable medium (also referred to as a computer-readable medium, a processor-readable medium, or a computer usable medium having a computer-readable program code embodied therein). The machine-readable medium can be any suitable tangible medium, including magnetic, optical, or electrical storage medium including a diskette, compact disk read only memory (CD-ROM), memory device (volatile or non-volatile), or similar storage mechanism. The machine-readable medium can contain various sets of instructions, code sequences, configuration information, or other data, which, when executed, cause a processor to perform steps in a method according to an embodiment of the invention. Those of ordinary skill in the art will appreciate that other instructions and operations necessary to implement the described invention can also be stored on the machine-readable medium. Software running from the machine-readable medium can interface with circuitry to perform the described tasks.

[0071] In particular, it should be appreciated that the source code can be developed on one computing apparatus, while the build time processes are executed by another apparatus, while protected application is actually executed on yet another. Each apparatus can comprise a processor, memory and a machine readable medium tangibly storing machine readable and executable instructions, which when executed by a processor, cause said processor to implement the methods disclosed herein.

[0072] The above-described embodiments of the invention are intended to be examples only. Alterations, modifications and variations can be effected to the particular embodiments by those of skill in the art without departing from the scope of the invention, which is defined solely by the claims appended hereto.

CLAIMS:

1. A method of transforming a binary software application comprising binary application code from an original form to a secured form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

A) performing a combination of a plurality of binary transmutations to said binary software application during a build time phase by making a series of changes to said binary application code to produce changed binary application code, said changes including implanting new code intertwined with said changed binary application code during build-time; and

B) interlocking said transmutations by generating and placing interdependencies between the transmutations;

C) during execution, applying said combination of transmutations and interlocking to both the binary application code to be protected and the implanted code; and

D) producing a protected application that is semantically equivalent to the original application but which comprises said interlocked transmutations such that the binary protection is no longer separated from the protected application.

2. The method of claim 2, wherein said combination of a plurality of binary transmutations comprises at least one inter-layer transmutation applied to successively nested layers of said binary application code.

3. The method of claim 2, wherein Step (B) comprises adding interlocking data, and wherein step (D) comprises producing a protected application which can not be executed correctly without the presence of valid interlocking data.

4. The method of claim 3, wherein Step (B) comprises having one transmutation require a previous transmutation to be present for proper execution.

5. The method of claim 4, wherein Step (B) comprises having one transmutation produce output that is used as input for a second transmutation.

6. The method of any one of claims 1-5 wherein Step (B) comprises using transmutations with complementary properties.

7. The method of any one of claims 2-6, wherein said successively nested layers comprise the modules comprising the application, the functions comprising a module, the basic blocks of instructions comprising the functions, and the individual machine instructions.

8. The method of any one of claims 2-7 wherein said combination of binary transmutations further comprises at least one intra-layer transmutation applied to a layer of said binary application code.

9. The method of 8 wherein said at least one intra-layer transmutation includes an integrity verification (IV) transmutation applied to at least one layer at both build time and run time.

10. The method of any one of claims 8 or 9 wherein said at least one intra-layer transmutation includes an anti-debug transmutation applied to at least one layer.

11. The method of any one of claims 2-10 wherein a binary transmutation comprises a static build time process comprising:

i) inserting transmutation execution code into the source code of the application to be protected to produce a changed source code, which is then compiled to produce said binary application code;

ii) inserting complimentary changes to the binary application to be protected such that the changes to the application source code do not work correctly unless said complementary changes to the binary application are present; and

iii) performing binary transmutation specific operations on the binary form of the application to be protected and generating associated static interlocking data that is required at execution time.

and wherein a binary transmutation further comprises a run time process comprising:

a. executing the transmutation execution code inserted into the application in such a manner that the original semantics of the application are not preserved if the transmutations are removed or modified;

b. dynamically generating run time interlocking data that is further required while

executing the transmutation execution code

c. utilizing said associated static and dynamic interlocking data to verify the interlocking dependencies as required;

d. executing a chain of multiple transmutations in a designated order such that one transmutation is required to generate an application state suitable for the execution of a second transmutation while executing the protected application.

12. The method of any one of claims 1-11 where a binary transmutation applies a white-box transformation process comprising :

at build-time:

i) utilizing a white-box transformation build-time facility to generate and assemble white-box transformation keys and operation code; and

ii) transforming said binary code and relevant important information from an input form to an output form by performing a white-box transformation operation with a white-box transformation build-time key, and;

iii) at run-time, utilizing a white-box transformation run-time facility to transform binary code and relevant important information from an output form back to an input form by performing a white-box transformation inverse operation with a white-box transformation run-time key.

13. The method of claim 12, wherein the white-box transformation build-time facility comprises:

a. a white-box transformation generator;
b. a white-box transformation build-time key master
c. a white-box transformation operation master; and
d. a white-box transformation build-time manager

and wherein the white-box transformation run-time facility comprises:

e. a white-box transformation run-time key master;
f. a white-box transformation inverse operation master; and
g. a white-box transformation run-time manager.

14. The method of claim 13, wherein the white-box transformation generator accepts original

key data and a transformation algorithm selection supplied by a user, and generates mated pairs of white-box transformation build-time key data (along with semantic operation code) coupled with the corresponding white-box transformation run-time key data (along with the inverse operation code).

15. The method of claim 14, further comprising: hiding the original key data and important transformation information after white-box transformation run-time key data and operation code are generated, to prevent inadvertent leaking while the corresponding white-box transformation inverse operation code is executing.

16. The method of claim 13 or 14, wherein multiple transformations are applied, and wherein said white-box transformation operation master stores multiple white-box transformation operations, said white-box transformation build-time key master stores multiple white-box transformation keys and wherein said white-box transformation run-time key master stores multiple white-box transformation run-time keys and said white-box transformation inverse operation master stores multiple white-box transformation inverse operations.

17. The method of claim 13, wherein:

- said white-box transformation build-time key master is accessed by the white-box transformation build-time manager;

- said white-box transformation operation master organizes and locates multiple white-box transformation semantic operation codes securely and effectively;

- said white-box transformation operation master is only accessed by the white-box transformation build-time manager;

- said white-box transformation build-time manager coordinates the storing, retrieving, and matching of various white box transformation build-time keys with their corresponding white box transformation semantic operation code;

- such that each white-box transformation process utilizes a white-box transformation build-time manager to invoke the proper white-box transformation semantic operations using the corresponding white-box transformation build-time keys at run-time

- said white-box transformation run-time key master organizes, stores and locates multiple white-box transformation run-time keys securely and effectively;

-said white-box transformation run-time key master is accessed by the white-box

transformation run-time manager;

- said white-box transformation inverse operation master organizes and locates multiple white-box transformation inverse operation code securely and effectively;

- said white-box transformation inverse operation master is accessed by the white-box transformation run-time manager;

- said white-box transformation run-time manager coordinates the storing, retrieving, and matching of various white box transformation run-time keys with their corresponding white box transformation inverse operation code; and

- such that each white-box transformation process utilizes a white-box transformation run-time manager to invoke the proper white-box transformation inverse operations using the corresponding white-box transformation run-time keys at run-time.

18. The method of claim 13, further comprising: selecting a wide range of computational algorithms including most popular cryptographic algorithms for white-box transformation algorithms implemented by said white-box transformation generator.

19. The method of claim 14 wherein, if a cryptographic algorithm is selected and a crypto key is provided by a user, the white-box transformation generator generates:

b. a white-box encrypt operation code as white-box transformation operation code

c. a white-box decrypt operation code as white-box transformation inverse operation code

d. a white-box encrypt key data as white-box transformation build-time key data

e. a white-box decrypt key data as white-box transformation run-time key data

20. The method of any one of claims 12-19, wherein different white-box transformation build-time and run-time keys and operation code are generated and assembled for different transmutations at different successively nested layers.

21. The method of any one of claims 2-20 wherein the method adds interlocking protection comprised of making changes to the source code that work in conjunction with changes made to the binary code in such a manner that both changes need to be present for the protected application to function correctly and wherein the interlocking comprises both inter-layer and intra-layer transmutations in such a manner as to make it difficult to remove any or all of the

transmutations, and wherein transmutations are interlocked by nesting transmutations, with each nested transmutation providing further protection to the layer it encapsulates, while receiving further protection from an encapsulating layer, and additionally including at least one intra-layer transmutation, with each intra-layer transmutation providing overlapping protection within either a layer of the protected application, or for the protected application as a whole.

22. The method of any one of claims 9-21 wherein the IV Transmutation is comprised of a IV processing at build-time comprising:

a. generating and assembling IV specific white-box transformation keys and operation codes;
b. computing static IV voucher data that represents hashing information of the application binary code at build-time;
c. applying a IV specific white-box transformation to transform said voucher data of said application by using said specific white-box transformation keys and operation codes to prevent un-authorized access and attacks to the voucher data;
d. assembling hidden white box transformation run-time key data corresponding to said specific white-box transformation keys and operation codes;

and wherein the IV Transmutation is further comprised of run-time actions at the beginning of and during the execution of the protected application comprising:

e. while an IV library is invoked, an IV initializer interacts with the white-box transformation run-time facility to inverse-transform the white-box transformed IV data to the plain IV data by executing the IV specific white box transformation inverse operation on white box transformation run-time key data and load the plain data into a protected data structure;
f. computing dynamic IV voucher data of the binary code loaded into memory by an OS, said dynamic IV voucher data representing hashing information of the application binary code at run-time, and storing the dynamic IV voucher data into a protected data structure;
g. checking the integrity of the application code to be protected by comparing said static IV voucher data, which is presented in protected form, against said dynamic IV voucher data.

23. The method of claim 22 wherein the application of the IV transmutation includes checks for integrity determined by user specification of designated options by allowing users inserting IV API calls into selected places of source code of the application during the build-time phase and subsequently automatically adding an automatic integrity verification engine to the application during build time which executes during run time.

24. The method of claim 22 or 23 wherein the IV transmutation checks the integrity of the binary application code itself and also checks the integrity of the code of any other binary transmutations applied to the code.

25. The method of any one of claims 21-24 wherein the IV transmutation execution is interlocked to the application by invoking callback functions.

26. The method of claim 25 wherein the callback functions are added during the build-time phase and are interlocked with IV transmutation execution through inserting IV API calls with callback functions as parameters during the IV transmutation preparation.

27. The method of any one of claims 12-26 wherein one of said binary transmutations comprises a Module Transmutation (MT) which protects an application module, which can be an application executable and/or dynamically shared library modules, by MT processing at build-time and at run-time in order to prevent static attacks to said application modules, wherein the MT processing at build-time comprises:

a. interacting with the white-box transformation build-time facility to generate and assemble MT specific white-box transformation keys and operations, wherein said assembled MT specific white-box transformation keys and operations comprise MT specific white-box transformation build-time key data and a white-box transformation operation along with corresponding white box transformation run-time key data and a white-box inverse operation ;
b. compressing the binary code of an application module to be protected to form compressed binary code;
c. applying said MT specific white-box transformation operation to transform said compressed binary code by using said MT specific white-box transformation

build-time key data and white-box transformation operation (from step a) to form a white-box transformed module ;

d. generating a securely packaged application module for said application module by combining said white-box transformed module (from step c) with a Resident Security Module (RSM) that contains said white box transformation run-time key data and inverse operation (assembled in step a) in hidden form; and

wherein the Module Transmutation (MT) is further comprised of run-time processing including:

e. while OS is loading the securely packaged application module and triggering the execution of code in the RSM, the RSM interacts with the white-box transformation run-time facility to inverse-transform the white-box transformed module to the compressed application module by executing the IV specific white box transformation inverse operation using the white box transformation run-time key data;

f. applying an uncompression operation to the compressed application module to obtain the binary code of said application module

g. mapping and loading said application module into memory, and transferring control to the entry point of the application.

28. The method of 27 wherein said RSM is protected by applying one or both of ADDand IV binary transmutations to said RSM.

29. The method of any one of claims 12-28 wherein one of said binary transmutations comprises at least one Function Transmutation (FT) which protects individual functions within application modules, by FT processing at build-time and at run-time in order to prevent static and dynamic attacks to said functions wherein a Function Transmutation is comprised of FT processing at build-time including:

a. interacting with the white-box transformation build-time facility to generate and assemble FT specific white-box transformation keys and operation codes unique to each of functions to be protected;

b. producing transformed functions by applying an FT specific white-box transformation to transform the binary code comprising each of the said functions by using white-box transformation build-time key data and white-box transformation operations specific to each of the functions (from step a);

c. installing entry and exit function handlers for each of said transformed functions so that all calls to each transformed function are performed via its entry function handler while return from the function is done via its exit function handler;

d. preparing a securely packaged application module for said application module comprising a Resident Security Module (RSM) and all protected functions by combining the white-box transformed functions with hidden white box transformation run-time key data and inverse operation assembled in step (a) corresponding to white-box transformation build-time key data and white-box transformation operation used on step (b), and all entry and exit function handlers installed in step (c) for all protected functions and

wherein the Function Transmutation (FT) is further comprised of run-time actions for each FT protected function including:

e. during the application module execution, when a protected function is invoked, its entry function handler is first executed and interacts with the white-box transformation run-time facility to inverse-transform the white-box transformed function code by executing the white box transformation inverse operation using the white box transformation run-time key data and operation codes specific to the function;

f. loading the inverse-transformed functions into execution memory; and

g. transferring the execution control to the function and, upon exit from the function, invoking the function's exit handler.

30. The method of claim 29 wherein said RSM, for each function, clears or scrambles the function's memory footprint before returning.

31. The method of claim 29 or 30 wherein intra layer protection is invoked such that one or both of an IV or ADB transmutation is interlocked with the FT applied to a function being protected by applying the IV, ADB or both transmutations to the entry and exit function handlers implicitly.

32. The method of any one of claims 29 -31 wherein an additional inter-layer transmutation, namely a Block Transmutation (BT) protects basic blocks of code within individual functions within application modules by BT processing at build-time and at run-time in order to provide

much stronger protection against static, dynamic and automatic attacks to said functions, and in a particular embodiment, wherein the Block Transmutation is comprised of BT processing at build-time including:

a. analyzing control flow and data flow of the said function to identify and determine block information and the structure of the function;
b. augmenting each block by installing entry and exit block handlers for each of blocks so that all execution paths reaching each of the blocks being protected invoke its entry block handler first, and then reach the block code, and finally leave from the block via its exit block handler accordingly.
c. interacting with the white-box transformation build-time facility to generate and assemble BT specific white-box transformation keys and operation codes unique to each of the blocks to be protected;
d. producing transformed blocks by applying a BT specific white-box transformation to transform the augmented binary code of each of the basic blocks (as processed in step b), by using white-box transformation build-time key data and white-box transformation operation specific to each of blocks (as generated and assembled in step c);
e. preparing a securely packaged application module for said application module comprising a Resident Security Module (RSM) containing said transformed blocks with said entry and exit block handlers, and said hidden white box transformation run-time key data and inverse operation assembled;

wherein the Block Transmutation (BT) is further comprised of run-time actions for all protected blocks of one particular FT protected function including:

f. during the application module execution, each time a protected function containing protected blocks is invoked, all blocks that belong to the same function are randomly permuted in memory.
g. each time a protected block is to execute, its entry block handler is invoked which then interacts with the white-box transformation run-time facility to inverse-transform the white-box transformed block by executing the white box transformation inverse operation using the white box transformation run-time key data and operation codes specific to the block;
h. loading the inverse-transformed block in a designated execution memory;
i. transferring execution control to the block loaded and, upon exit from the block,

the exit block handler (optionally) clears or scrambles the memory footprint and transfers execution to the next block. Repeat step B, C and D until the protected function exits.

5 33. The method of claim 32, wherein the Block Transmutation (BT) further comprises random *code position* permutation of dynamically inverse-transformed blocks in memory based on the f*requency* of block permutation specified by users via the input security options.

34. The method of claim 32, wherein the Block Transmutation (BT) is further comprised that
0 *the exit block handler* can optionally interact with the white-box transformation run-time facility to re-transform the white-box inverse-transformed block by executing the white box transformation operation using the white box transformation run-time key data and operation codes specific to the block.

5 35. The method of claim 33 or 34 wherein the block transmutations are further protected by applying one or both of IV and/or ADB transmutations to each block's entry and exit block handlers.

36. The method of any one of claims 27-35 wherein the Instruction Transmutation (IT)
0 protects single instructions of code within individual functions within application modules, by IT processing at build-time and at run-time in order to provide much strong protections against static, dynamic and automatic attacks to said functions.

37. A system of protecting a software application comprising binary code and optionally
5 associated data, from an original form to a more secure form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said system comprising:

a. providing secure binary libraries for a software application to invoke the designated transmutation execution behaviors at code locations that user want to
0 protect.
b. providing options for users to apply different types of the transmutations at varied granularities, such as module, function, block and instruction that comprise said application.

c. providing a build-time toolset to perform binary transmutation preparations to the said application, and transform the original execution of the said application to a secured execution by using the toolset.

d. producing a protected application that is semantically equivalent to the original application, and comprises said interlocked transmutation executions, which also are interlocked with transmutation preparations, such that the binary protection is no longer separated from the protected application.

e. during executing a protected application, the secured execution of the said protected application comprising interlocked transmutation executions protects the execution such that prevents from attacks statically and dynamically and only very small portion of binary code presented in execution memory is in clear form during any time of its execution.

38. The system of claim 37, wherein c., d. and e. further comprise:

generating, arranging and adding interlocking data and code by interlocking between transmutation preparations and executions, and producing a protected application which can not be executed correctly without the presence of valid interlocking data and code;

generating, arranging and adding interlocking data and code by interlocking among executions of different transmutations, and producing a protected application which can not be executed correctly without the presence of valid interlocking data and code;

having multiple transmutations produce multiple layers of protections to the said application; and

wherein b. further includes overlapping more than one transmutation on at least one granularity.

39. The system of claim 38 further comprising having nested complimentary transmutations produce multiple nested layers of protections with complementary properties, each protected by IV, ADB or both.

40. The system of claim 39 further comprising splitting a granularity in parts and calling transmutation execution for a next part of the granularity only when an IV check result is determined to be successful on the current transmutation on the current granularity.

41. The system of any of claims 37-40 further comprising providing fail and success callback functionalities between the transmutations wherein white-box transformation inverse operation of a transmutation on a given granularity can only proceed if white-box transformation inverse operation of another transmutation succeeds.

42. The Method/System of any proceeding claim where said transmutations are applied both to said code and to associated data.

43. A computer program product comprising machine readable medium tangibly storing machine readable and executable instructions, which when executed by a processor, cause said processor to implement any of the methods disclosed and/or claimed herein.

44. The system of claim 37 wherein the white box transformation allows splitting a white box run-time key data into two or more parts, where at least one part is internal and embedded within the protected application, and the other parts are external and stored in one or more separate storage forms.

45. The system and method of claim 44, further comprising: allowing external white box transformation run-time key parts being provisioned separately to a device where the protected application, with the internal white box transformation run-time key part, is installed during a device provisioning time.

46. The system and method of claim 44, further comprising: providing a specific white box run-time operation invoked by the white box transformation run-time key master to reconstitute different and separated white box run-time key data parts into a usable form.

47. The system and method of claim 44, further comprising: creating an interlock dependency between the protected application and these separately provisioned external parts of the white box transformation run-time key to improve security as follows:

   a. an attacker must break multiple parts of the white-box key in order to get the whole key so that will make such an attack much harder and reduce security risk during any period of distributing and provisioning the protected application and external key information.

b. the implementer can introduce diversity by changing the ratio of multiple different parts of the key since different instances of the protected application and white-box transformation run-time key can have different percentages of key data splitting.

c. interlocking an instance of a protected application with one device or interlocking instances of a protected application with a set of devices that share the same key data format so that an instance of protected application to a single device whereas code lifting of all or part of the semantically equivalent application from the device fails to function on any unauthorized and matched devices.

48. The system and method of claim 47 where provisioning takes the form of:

d. install the external parts of the white-box transformation run-time key on a device by the manufacturer at manufacturing time or service provider at provisioning time;

e. downloading the external parts of the white-box transformation run-time key via a network when the application is invoked either the first time or every time;

f. obtaining the external part of the white-box transformation run-time key from a user-supplied device such as a smart card connected to the device when the application is invoked;

g. manually entering the external part of the white-box transformation run-time key by the user when the application is invoked.

49. A method of transforming a binary software application comprising binary application code from an original form to a secured form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

analyzing said binary application to determine at least one component of said application to which at least one binary transmutation can be applied, said component including component code;

performing a series of changes to said component code to produce changed component code, said changes including applying at least one WB transformation to said component code and implanting new code intertwined with said changes to said binary application code;

interlocking said changes by generating and placing interdependencies between said changes; and

applying said changes and interlocking to both the binary application code to be protected and the implanted code to produce a transmuted application that is semantically equivalent to the original application but which comprises said interlocked transformations such that the binary protection is no longer separated from the protected application.

50. A method of protecting a software application comprising binary code and optionally associated data, from an original form to a more secure form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

c. providing a build-time toolset to perform binary transmutation preparations to said application and transform the original execution of the said application to a secured execution by using the toolset; amd
d. producing a protected application that is semantically equivalent to the original application, comprising interlocked transmutation executions, which also are interlocked with transmutation preparations, such that the binary protection is no longer separated from the protected application;

wherein

the secured execution of the said protected application comprises interlocked transmutation executions configured such that only a small portion of binary code is in clear form during any time of its execution.

51. The method of claim 50, wherein step a comprises:

generating, arranging and adding interlocking data and code by interlocking between transmutation preparations and executions, and producing a protected application which can not be executed correctly without the presence of valid interlocking data and code;

52. The method of claim 51 wherein said generating, arranging and adding interlocking data and code comprises interlocking among executions of different transmutations of different granularities, and producing a protected application which can not be executed correctly without the presence of valid interlocking data and code; and

having multiple transmutations produce multiple layers of protections to the said application.

53. The method of claim 52 wherein said different transmutations of different granularities comprise having nested complimentary transmutations produce multiple nested layers of protections with complementary properties, each protected by IV, ADB or both.

54. The method of any one of claims 50- 53 further comprising:

a. providing secure binary libraries for a software application to invoke the designated transmutation execution behaviors at code locations that a user wants to protect; and
b. providing options for users to apply different types of the transmutations at varied granularities, such as module, function, block and instruction that comprise said application.

55. A method of transforming a binary software application comprising binary application code from an original form to a secured form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

A) performing a combination of a plurality of binary transformations to said binary software application during a build time phase by making a series of changes to said binary application code to produce changed binary application code, said changes including implanting new code intertwined with said changed binary application code during build-time;

B) interlocking said transformations by generating and placing interdependencies between the transformations; and

C) applying said combination of transformations and interlocking to both the binary application code to be protected and the implanted code to produce a transmuted application that is semantically equivalent to the original application but which comprises said interlocked transformations such that the binary protection is no longer separated from the protected application.

56. The method of claim 55, wherein: STEP A+B comprise performing binary transmutation

specific operations on the binary form of the application to be protected and generating associated static interlocking data that is required at execution time.

and wherein a binary transmutation further comprises a run time process comprising:

a. executing the transmutation execution code inserted into the application in such a manner that the original semantics of the application are not preserved if the transmutations are removed or modified;

b. dynamically generating run time interlocking data that is further required while executing the transmutation execution code

c. utilizing said associated static and dynamic interlocking data to verify the interlocking dependencies as required;

d. executing a chain of multiple transmutations in a designated order such that one transmutation is required to generate an application state suitable for the execution of a second transmutation while executing the protected application.

57. The method of claim 56 such that any given transmutation includes the insertion of the IV and ADB routines with their accompanying interlock facilities and the decomposing of the program into functions, blocks, and instructions that are then transformed with WB technology require a specific environment in which to run (which includes entry and exit handlers, an RSM, etc.) and is thus rendered practically irreversible.

58. A method of transforming a binary software application comprising binary application code from an original form to a secured form that is resistant to static and/or dynamic attacks attempting to tamper with, reverse engineer, or lift all or part of the application, said method comprising:

analyzing said binary application to determine components of said application to which binary transmutations can be applied, wherein each binary transmutation comprises:

performing changes to said code by apply a transformation to said component code and additionally implanting transmutation execution code intertwined with said

transformed component code;

interlocking said changes by generating and placing interdependencies between said changes and other aspects of said application code <which can include the main application, or other components >; and

applying said changes and interlocking to both the binary application code to be protected and the implanted code to produce a transmuted application that is semantically equivalent to the original application but which comprises said interlocked transformations such that the binary protection is no longer separated from the protected application.

59. The method as claimed in claim 58 wherein a binary transmutation comprises generating associated static interlocking data that is required at execution time.

and wherein a binary transmutation further comprises a run time process comprising:

a. executing the transmutation execution code inserted into the application in such a manner that the original semantics of the application are not preserved if the transmutations are removed or modified;

b. dynamically generating run time interlocking data that is further required while executing the transmutation execution code

c. utilizing said associated static and dynamic interlocking data to verify the interlocking dependencies as required;

d. executing a chain of multiple transmutations in a designated order such that one transmutation is required to generate an application state suitable for the execution of a second transmutation while executing the protected application.

60. A method as claimed in any one of the proceeding claims, further comprising Dynamic Function loading.

61. A method of dynamic function loading as described.

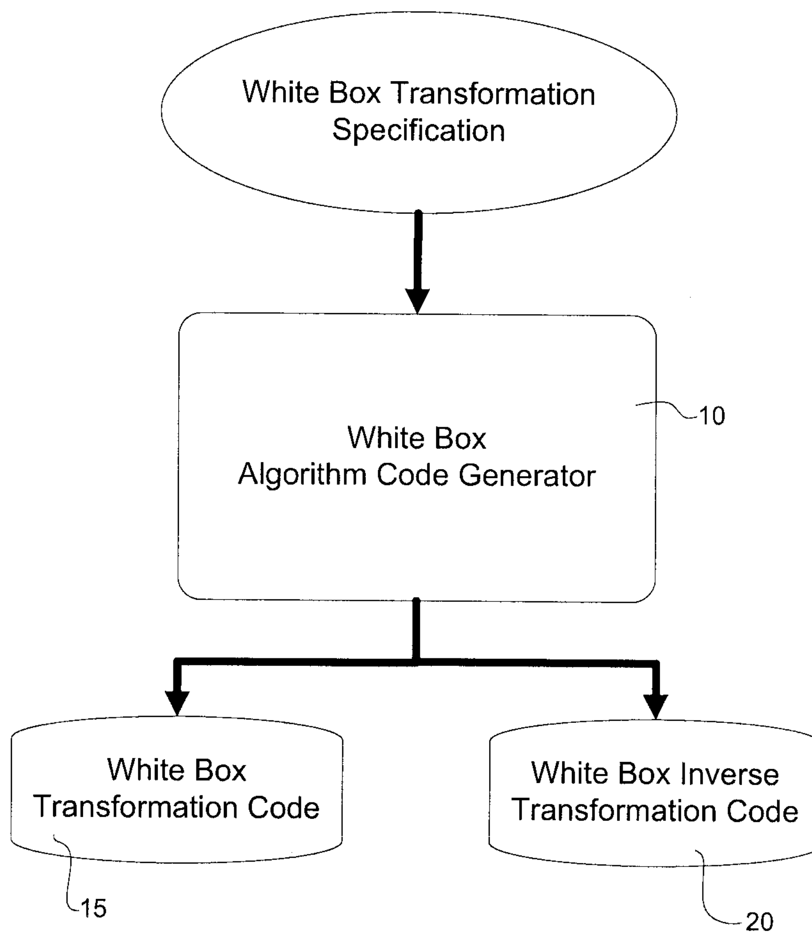


Figure 1

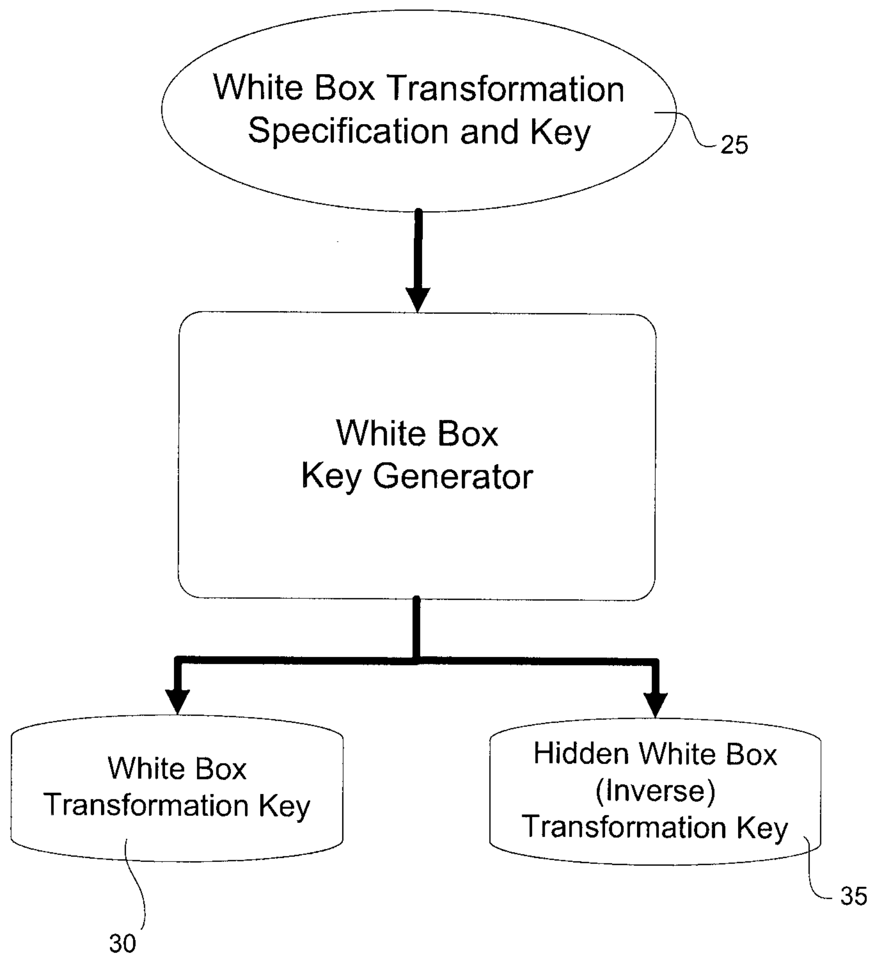


Figure 2

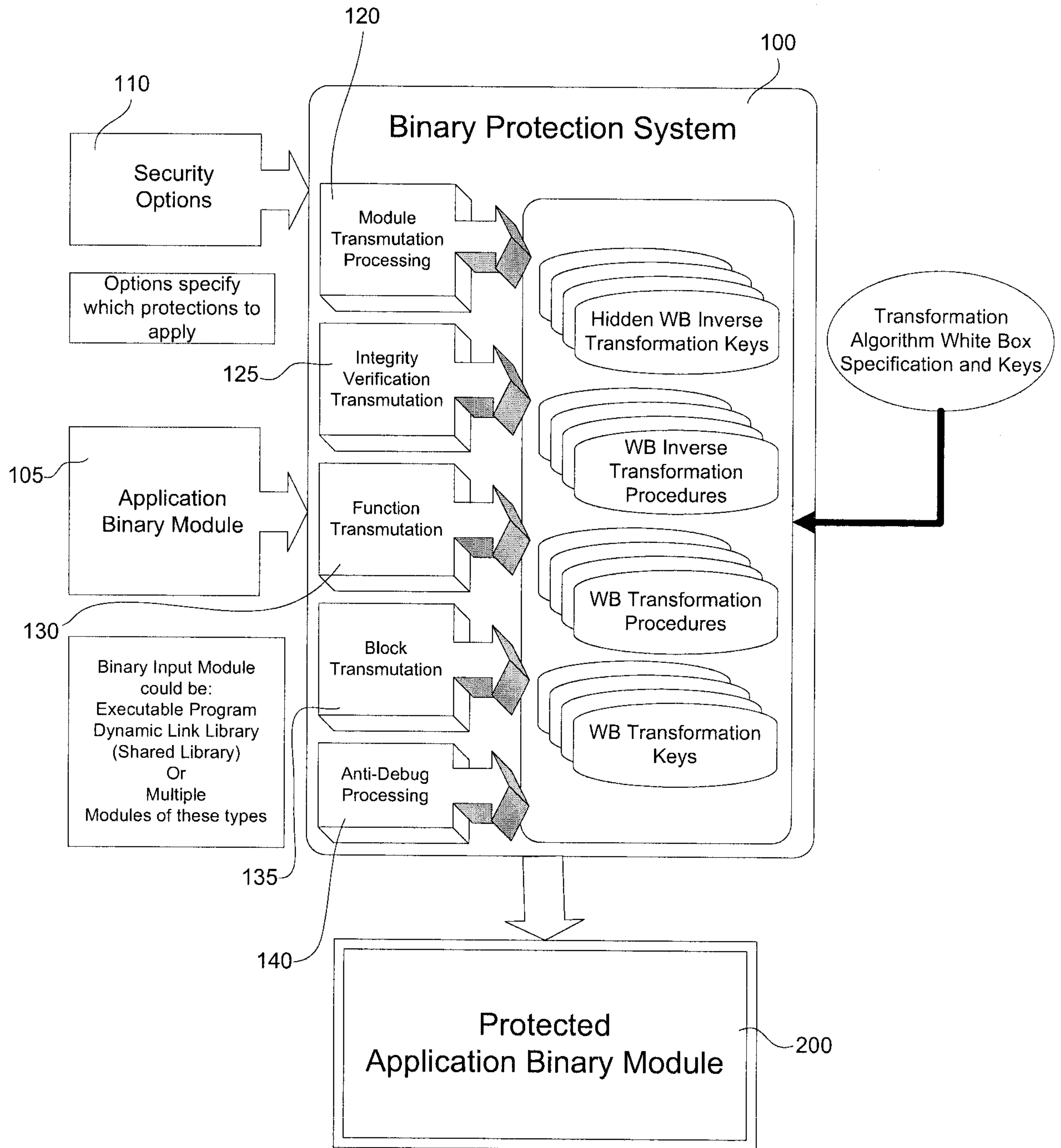


Figure 3

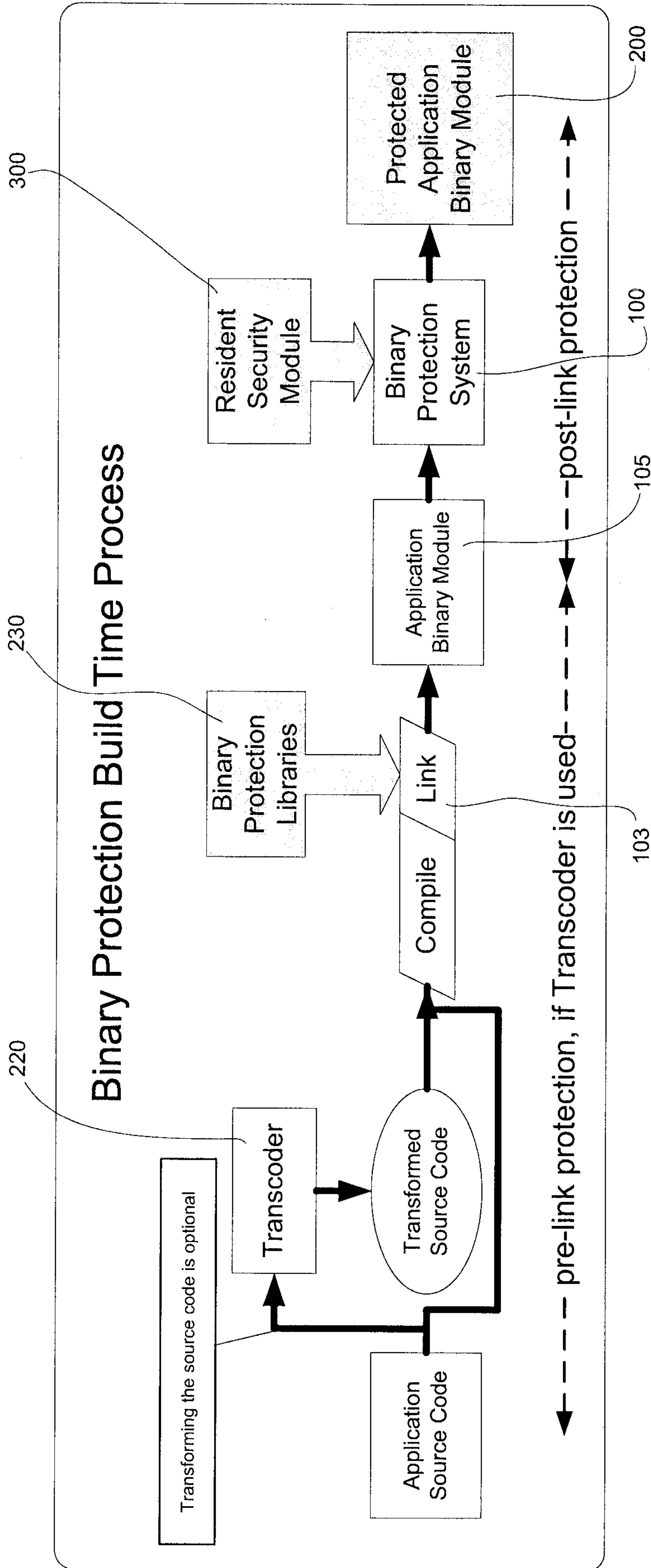


Figure 4

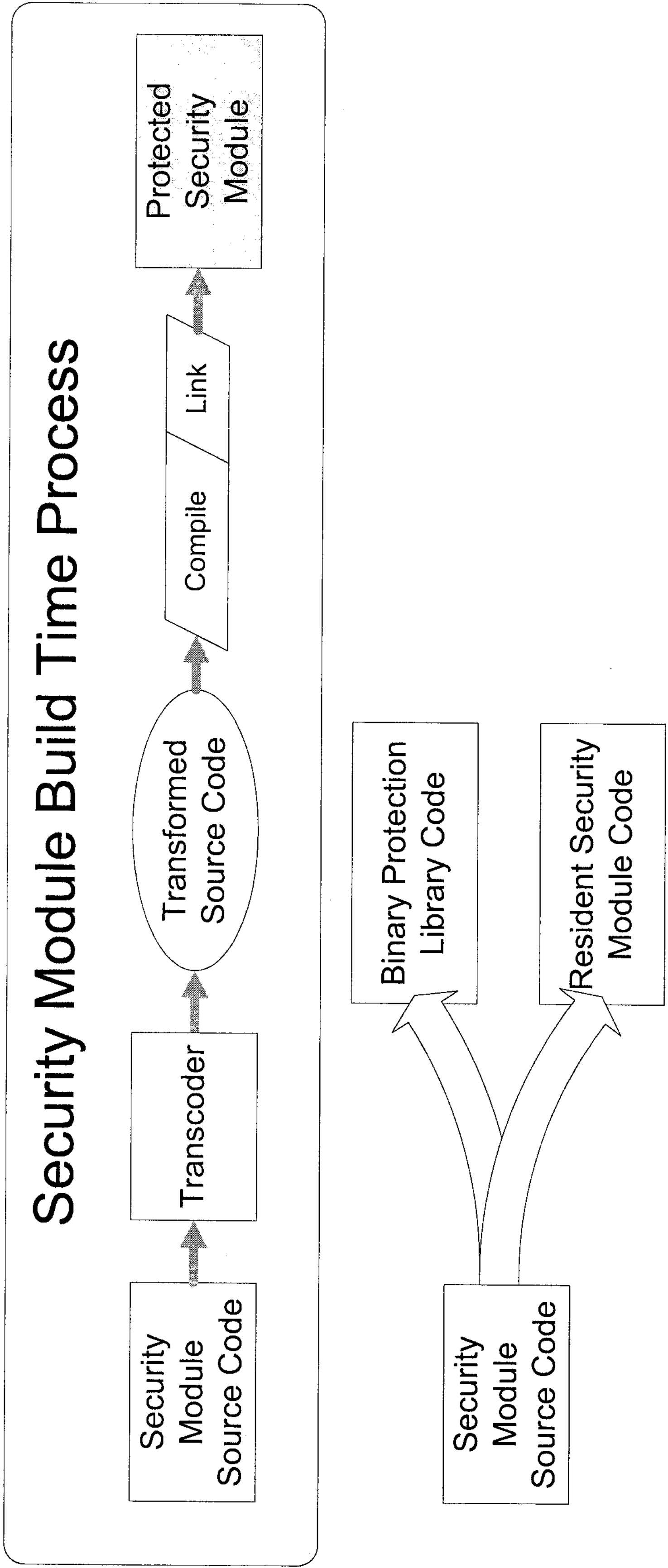


Figure 5

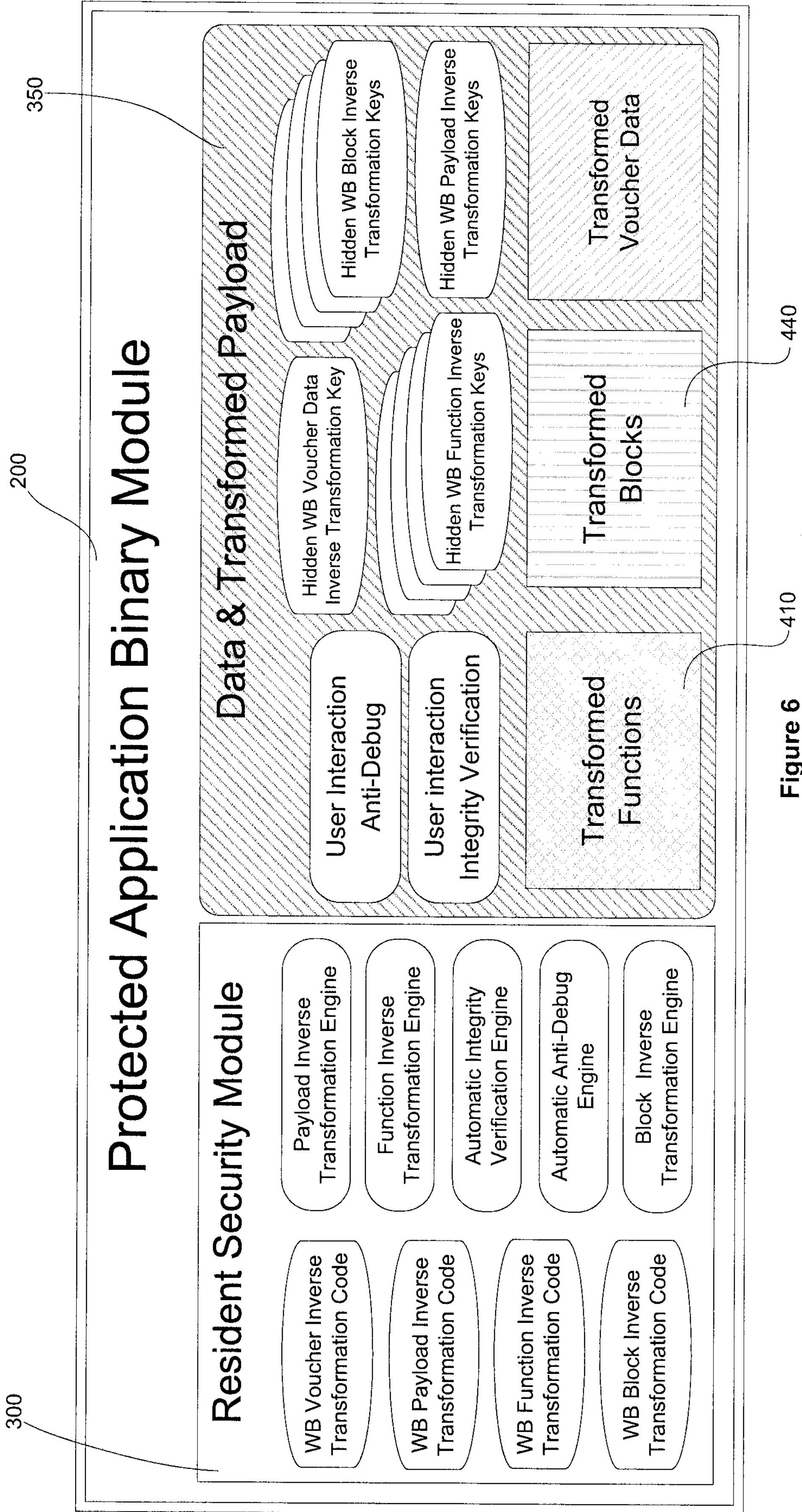


Figure 6

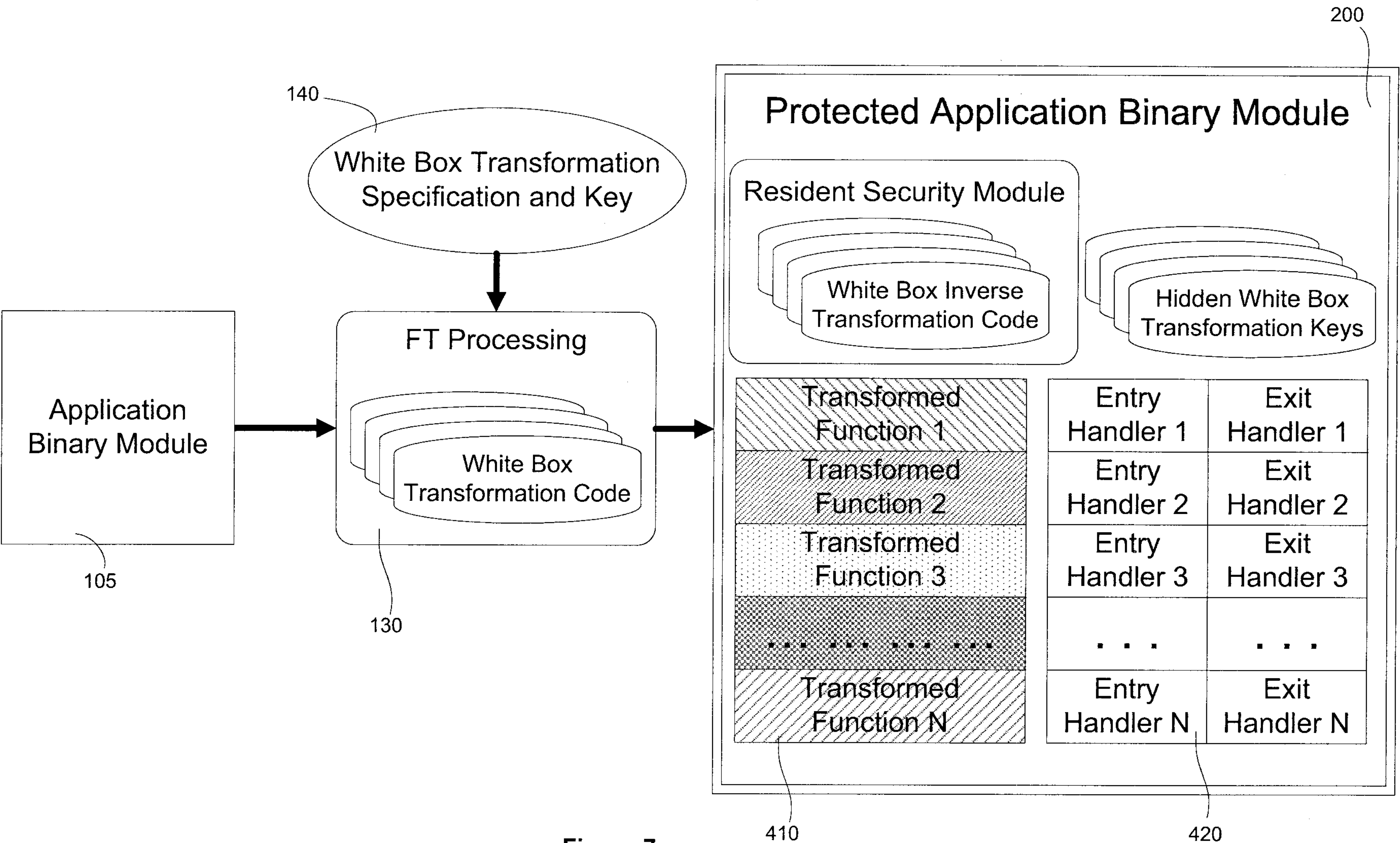


Figure 7

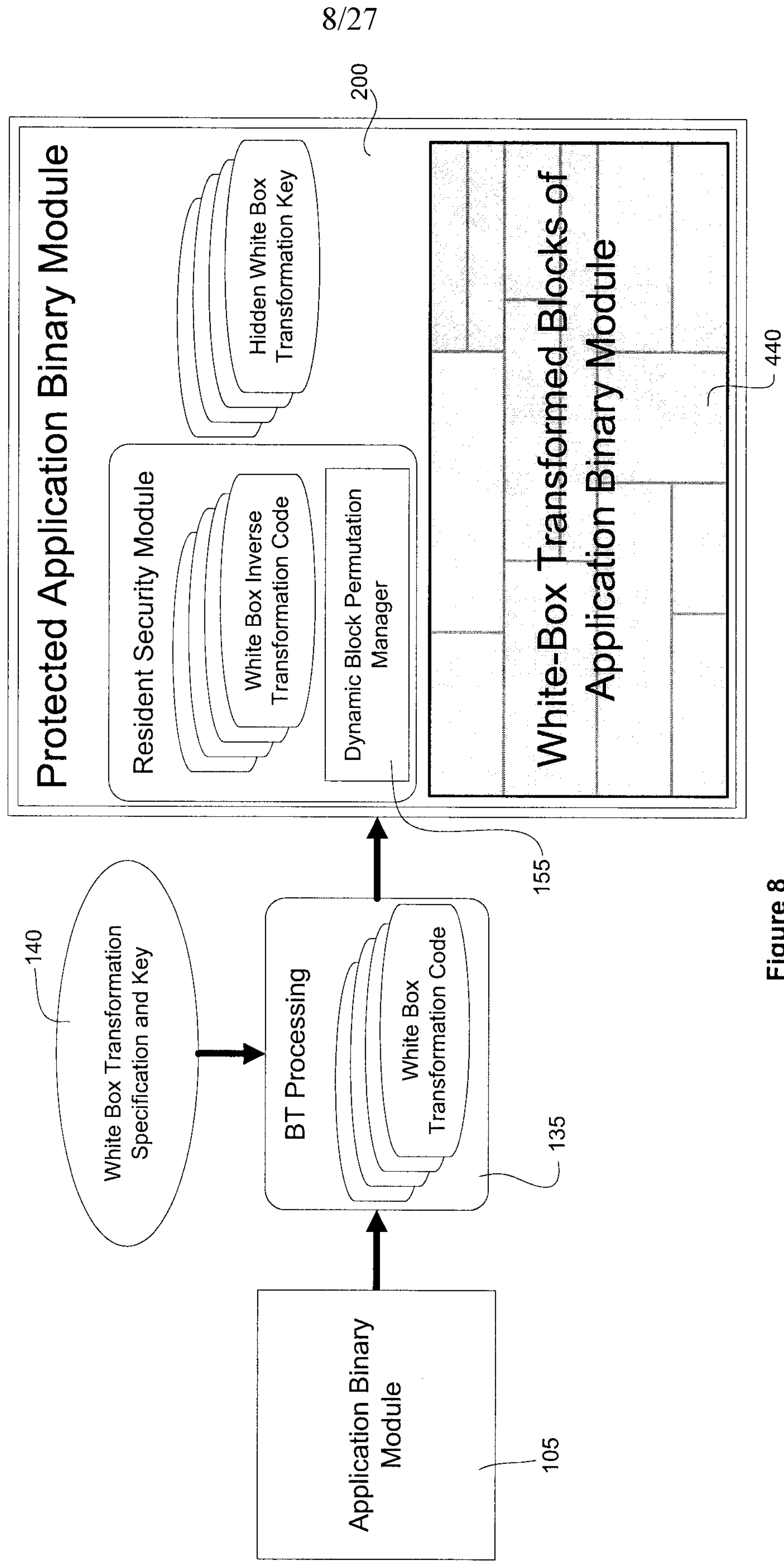


Figure 8

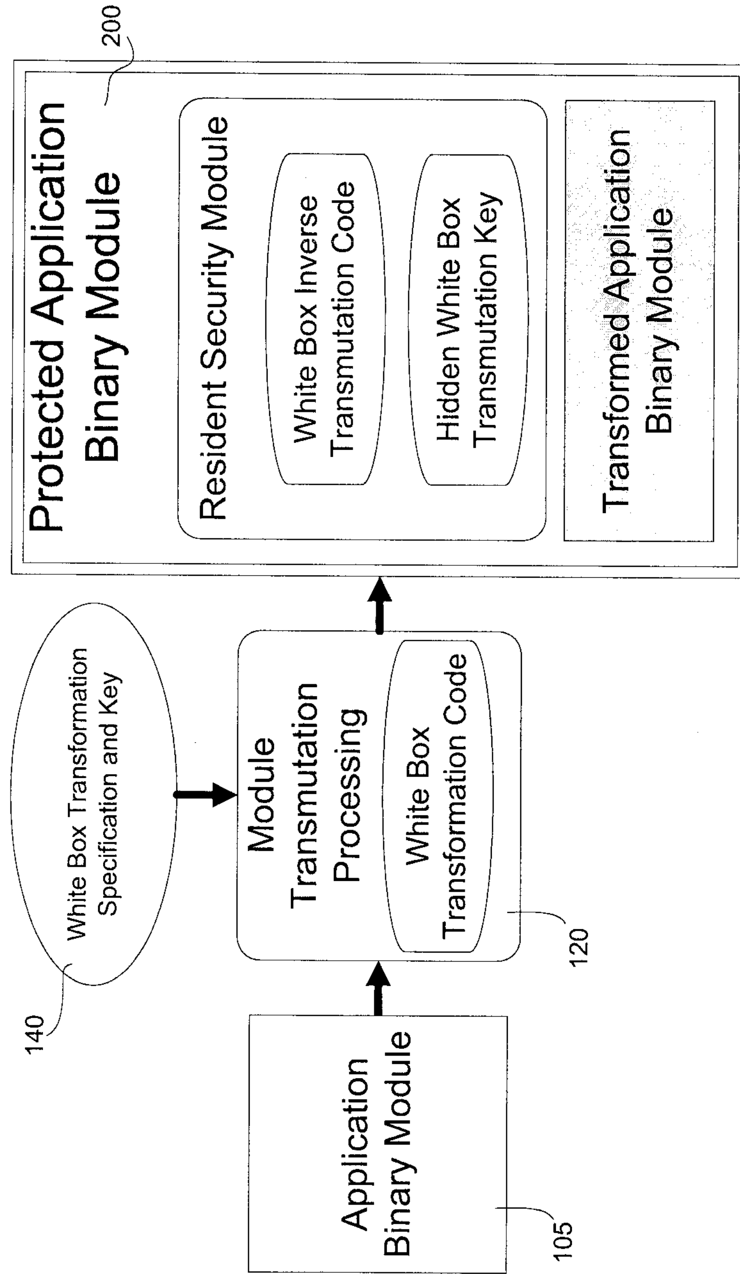


Figure 9

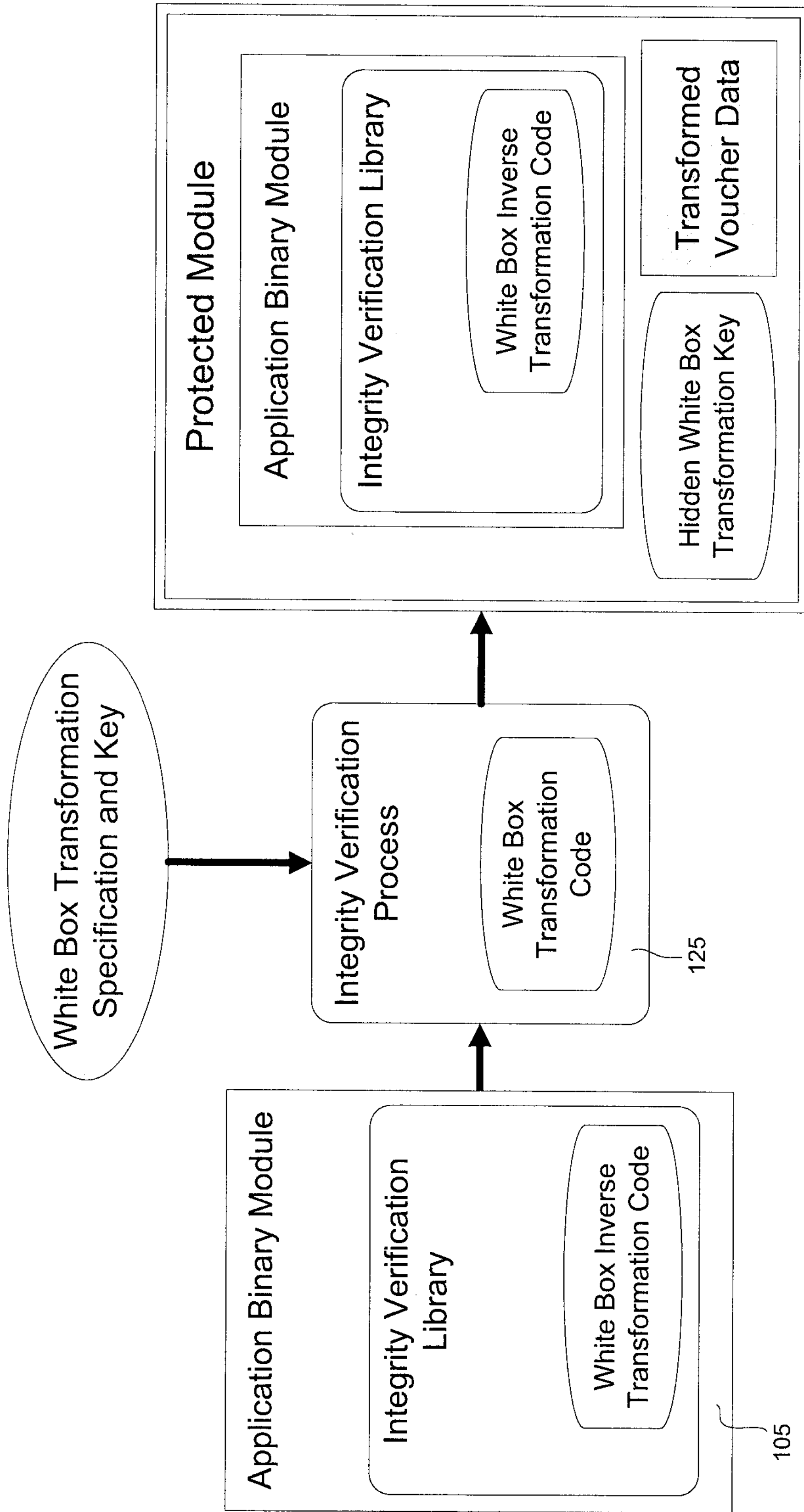


Figure 10

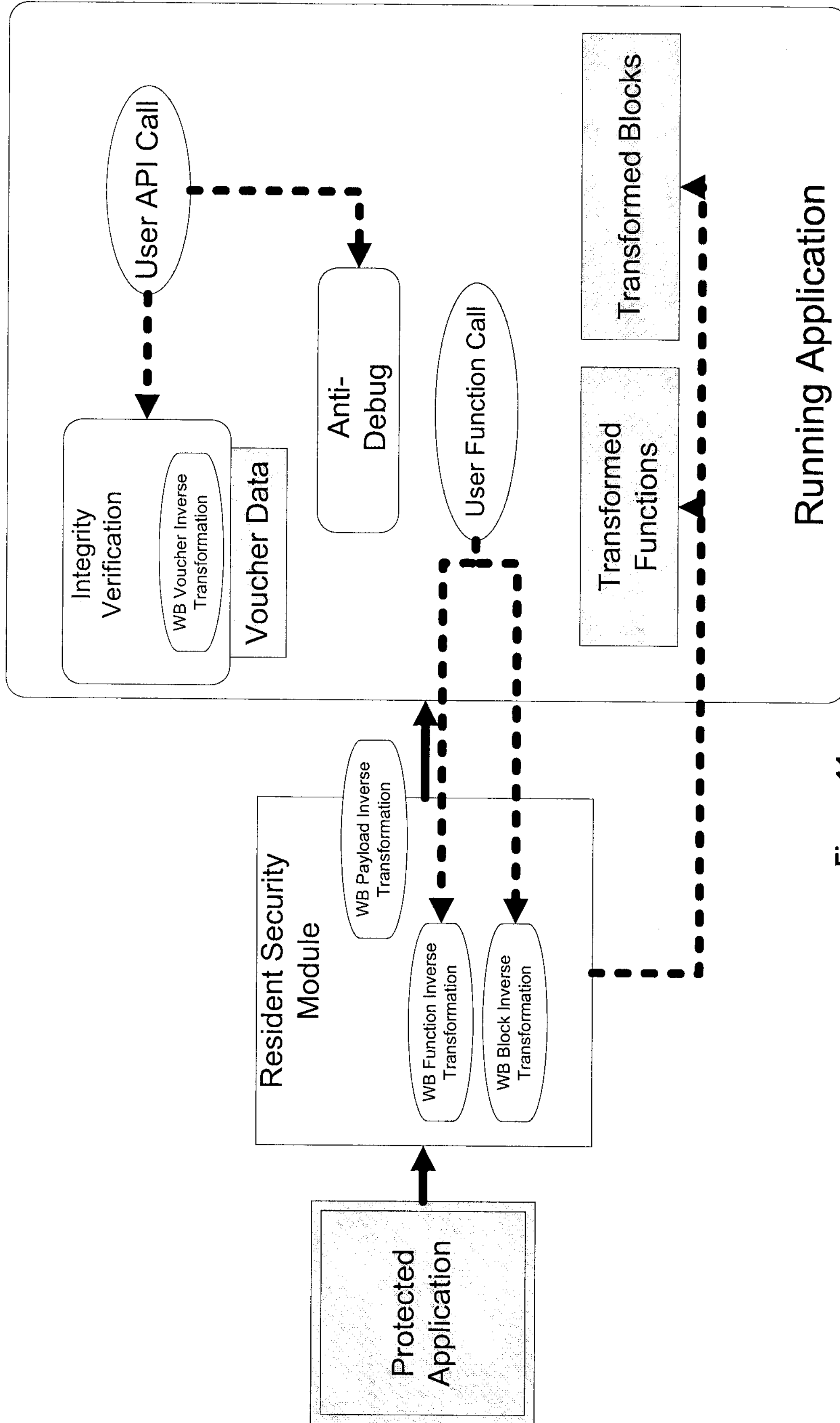


Figure 11

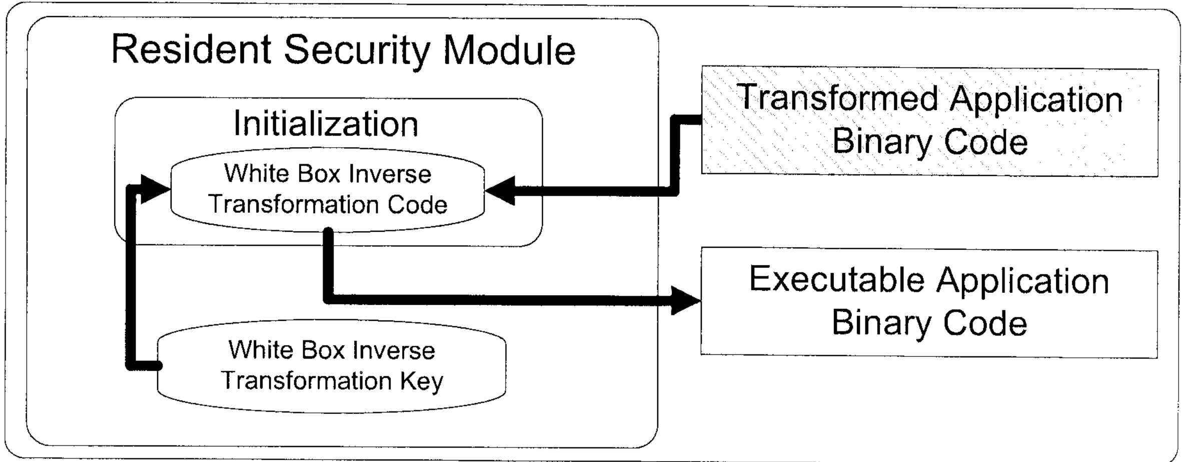


Figure 12

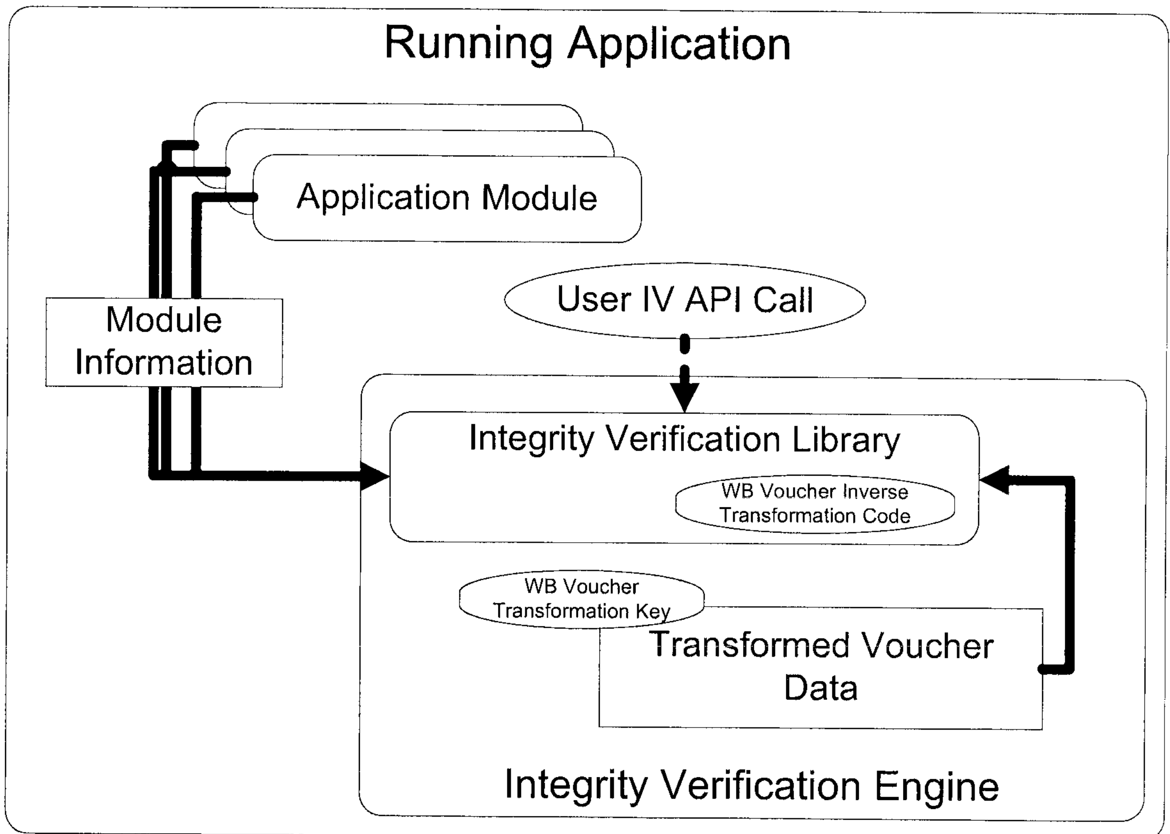


Figure 13

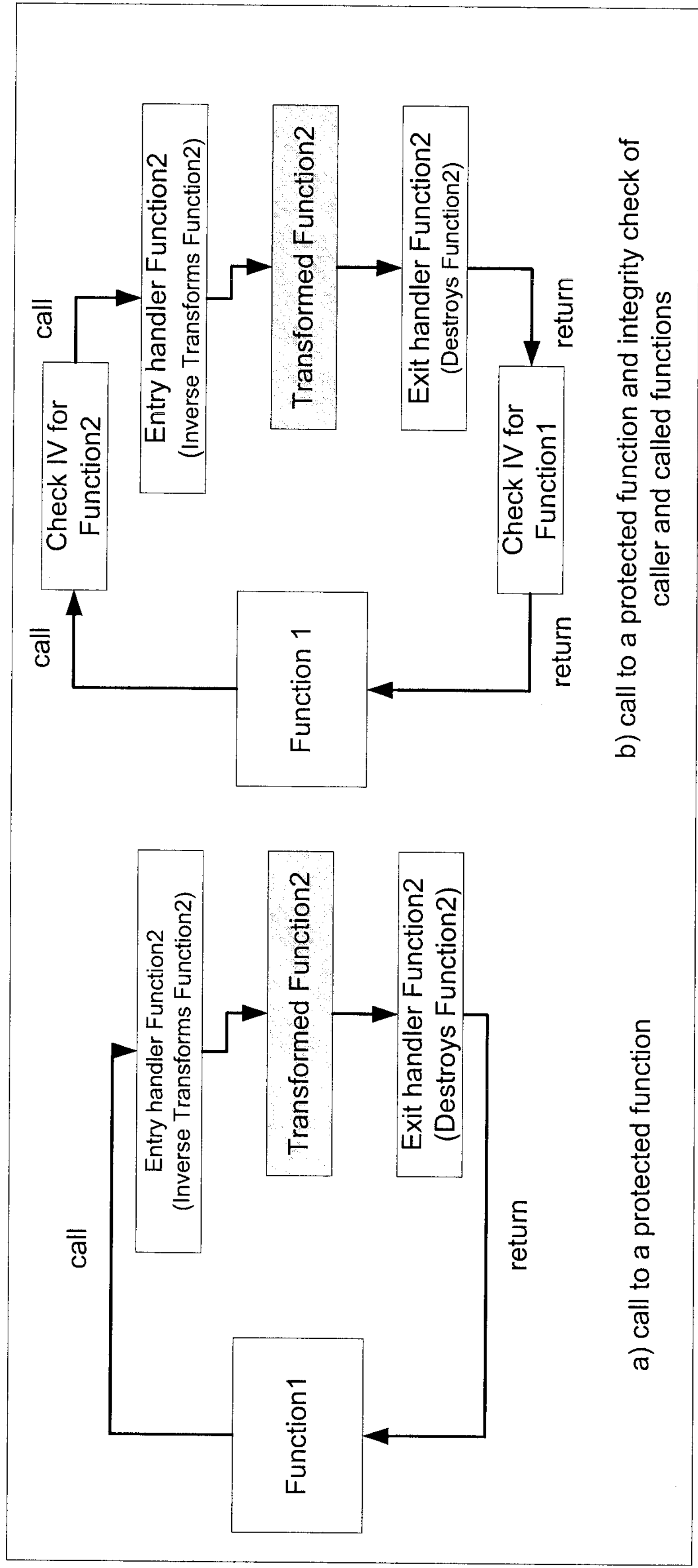


Figure 14

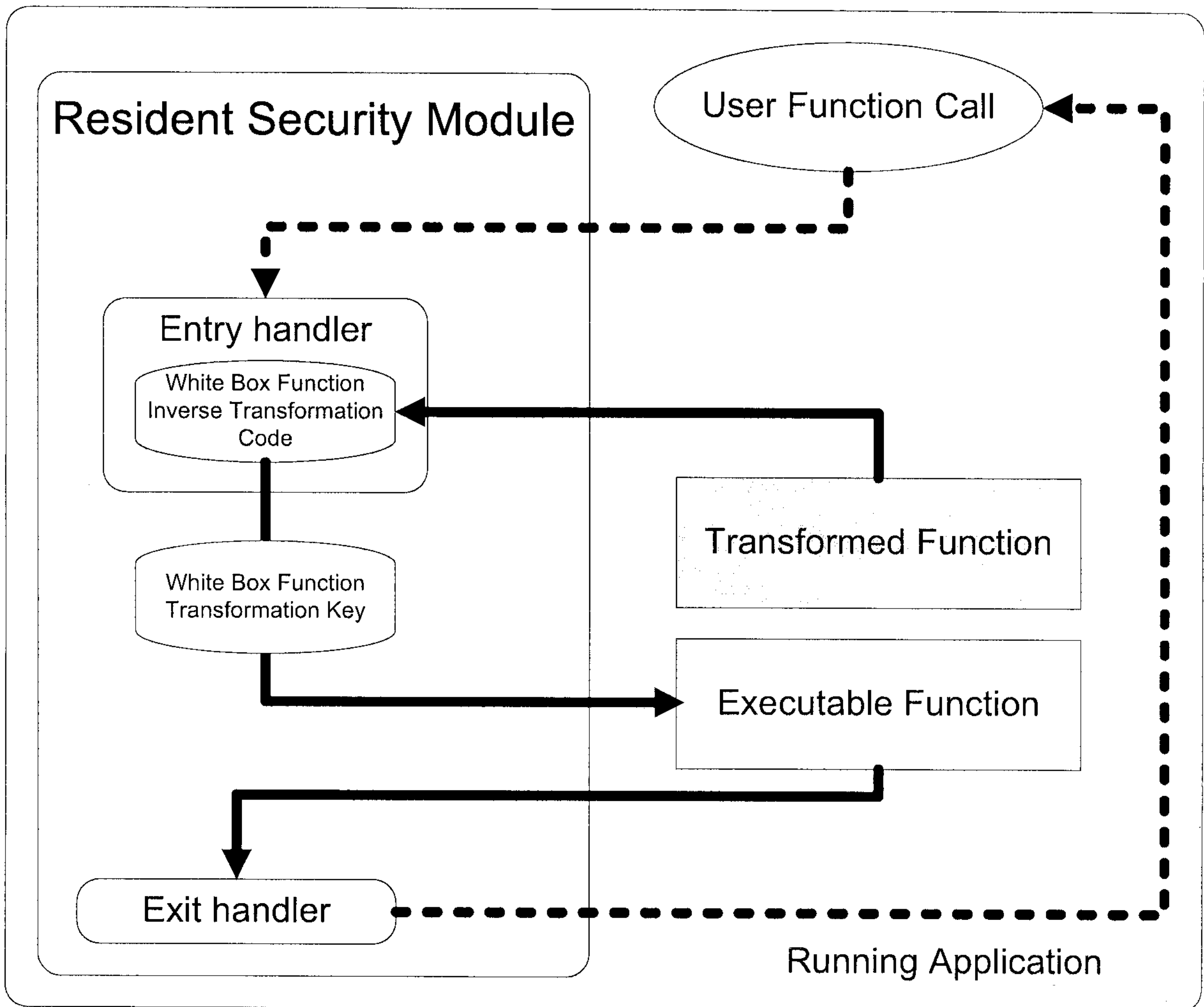


Figure 15

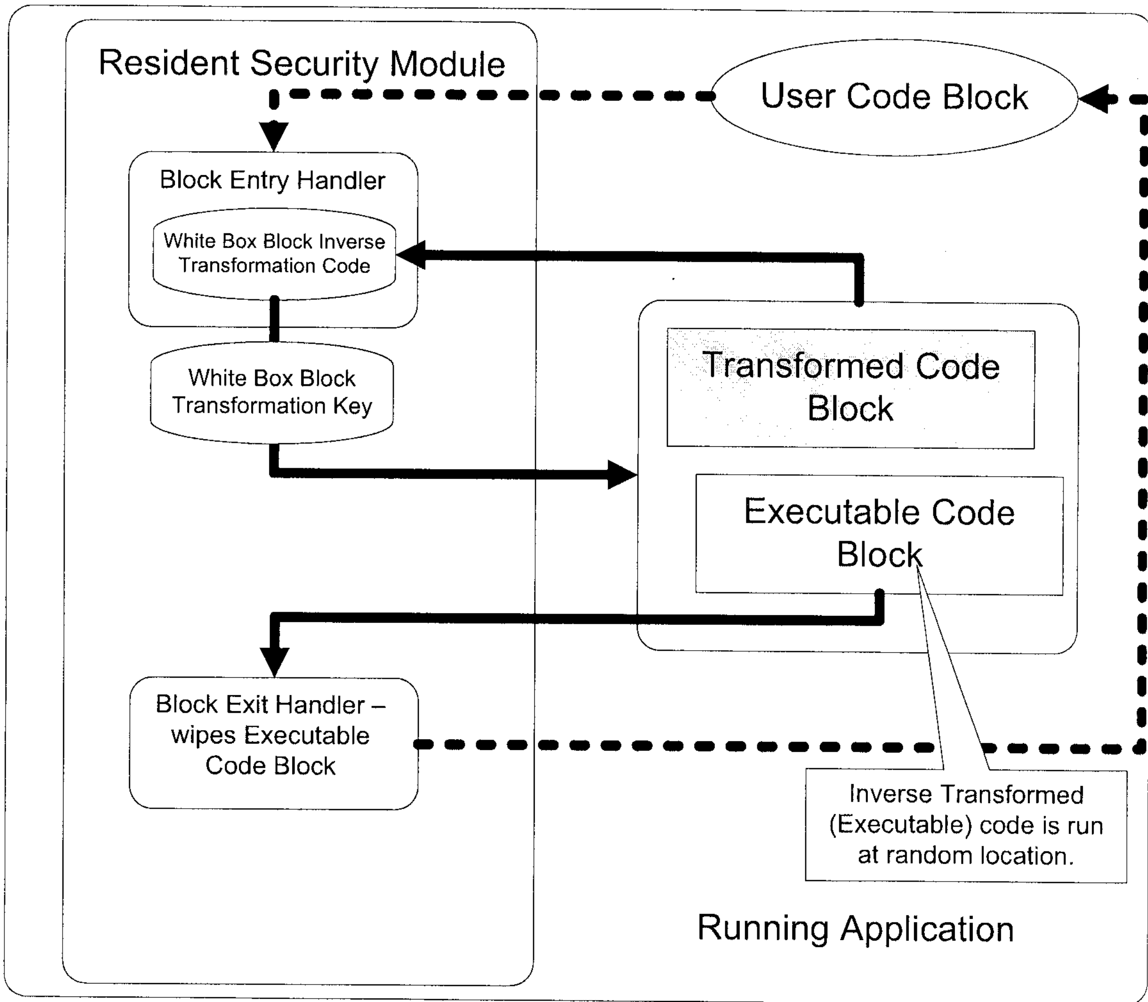


Figure 16

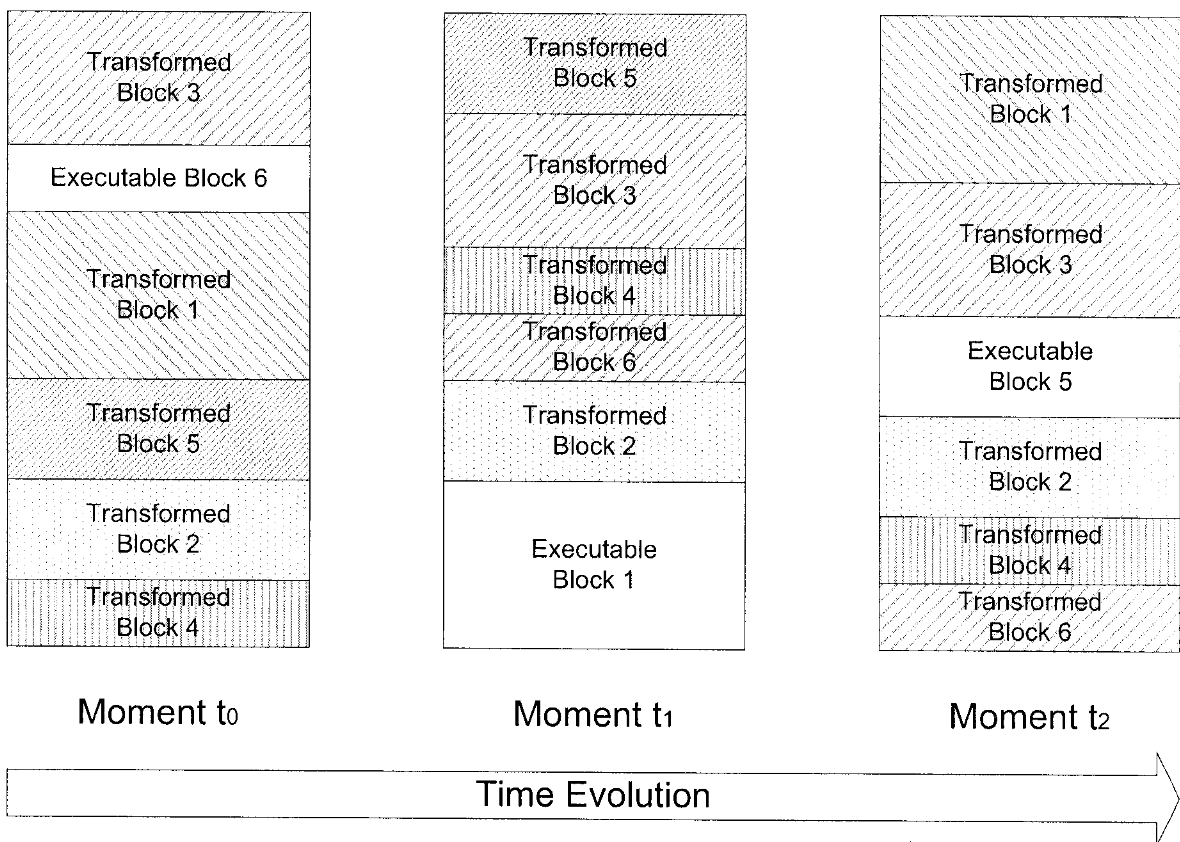


Figure 17

Dynamic Function Loading Build Time Process

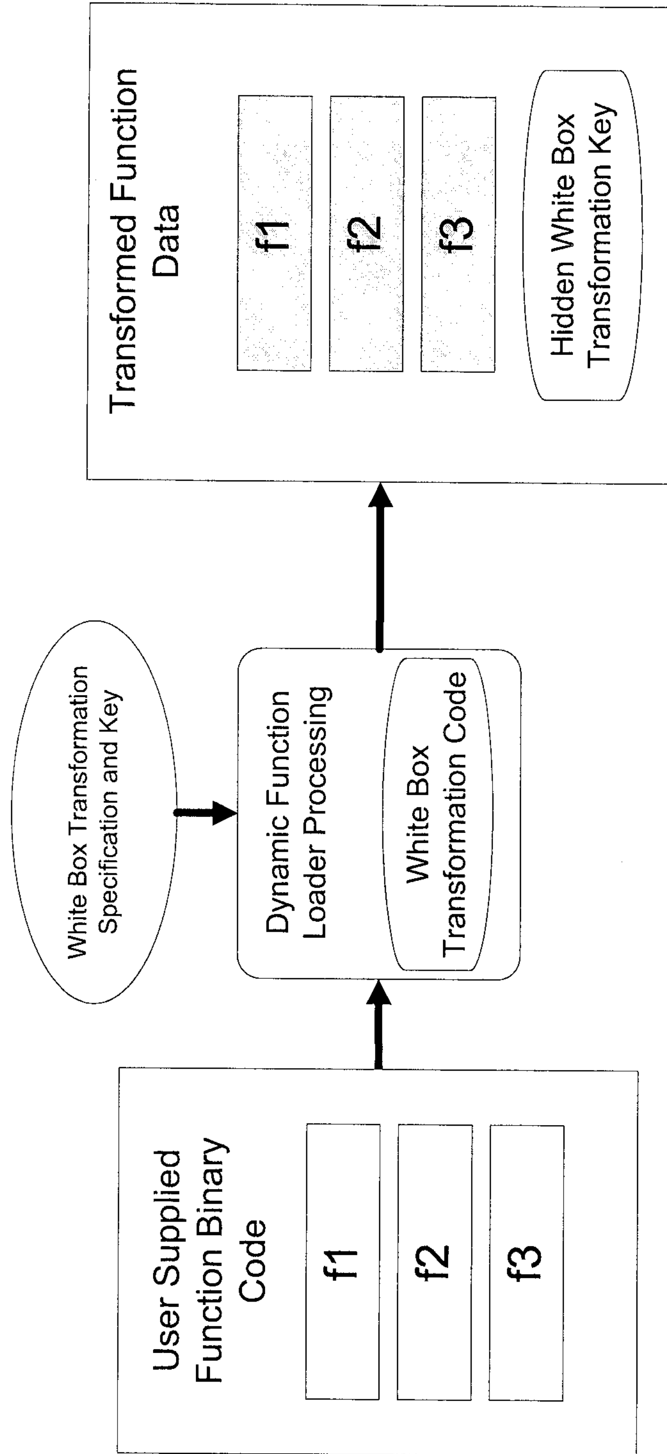


Figure 18

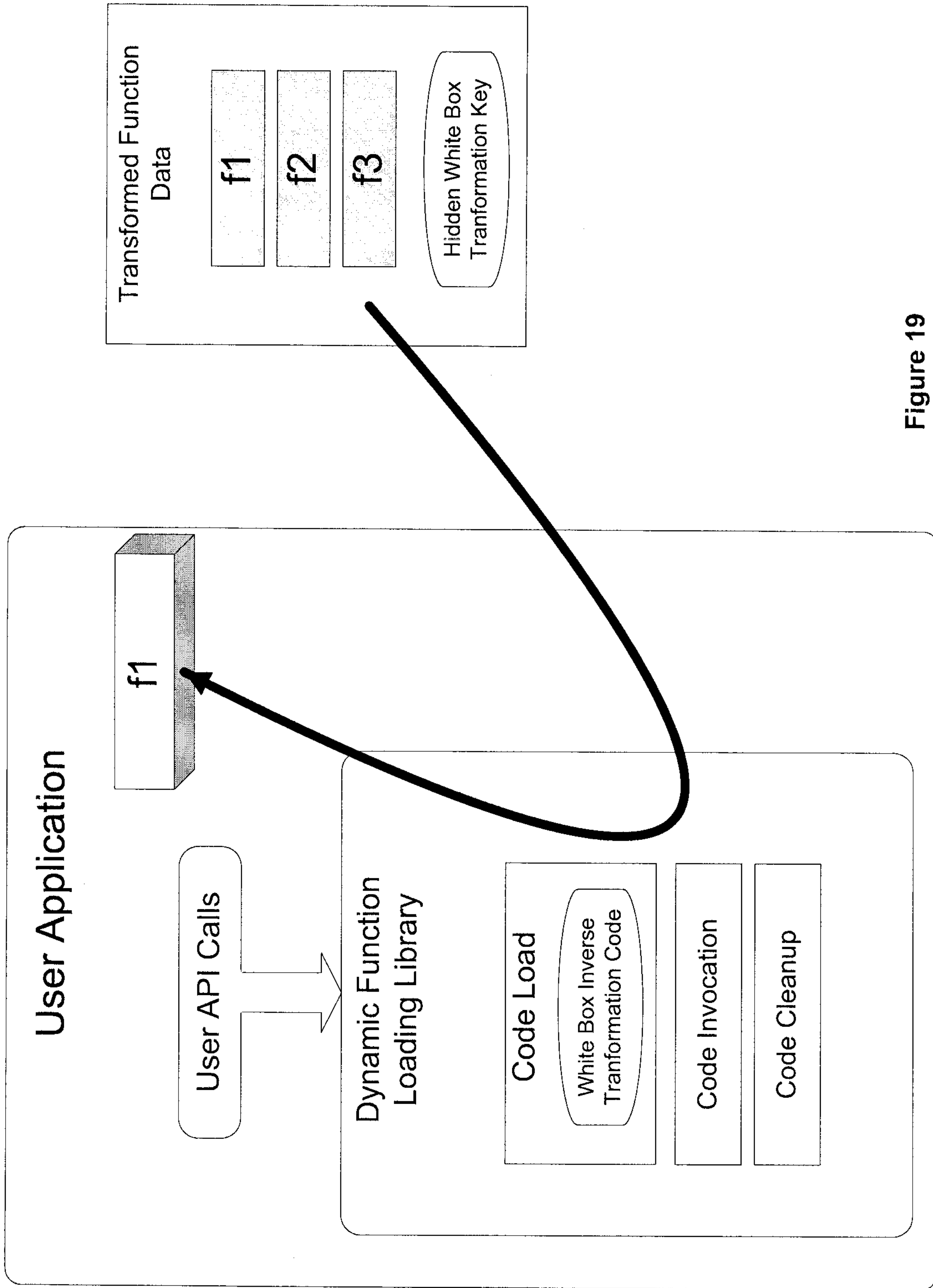


Figure 19

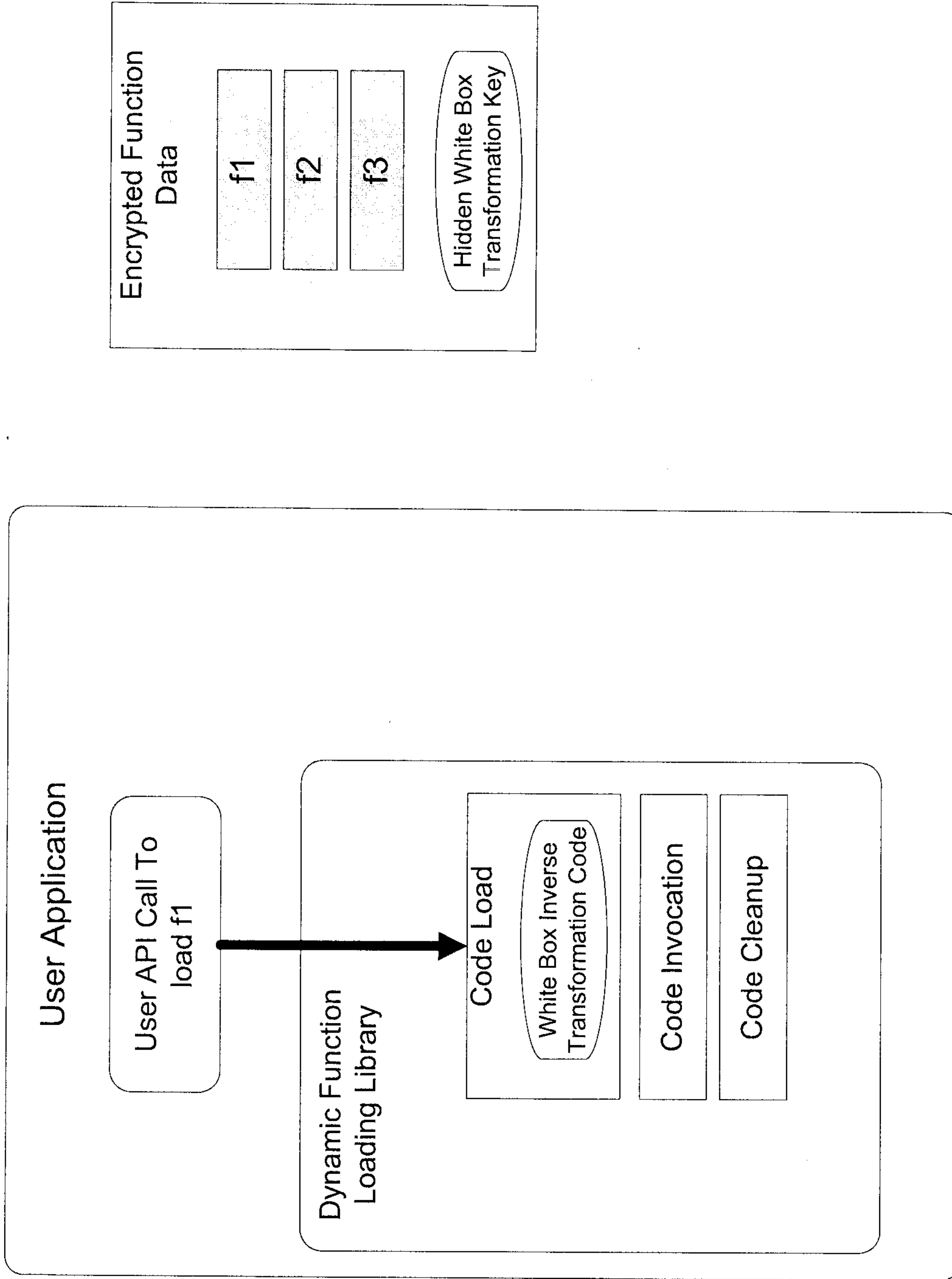


Figure 20

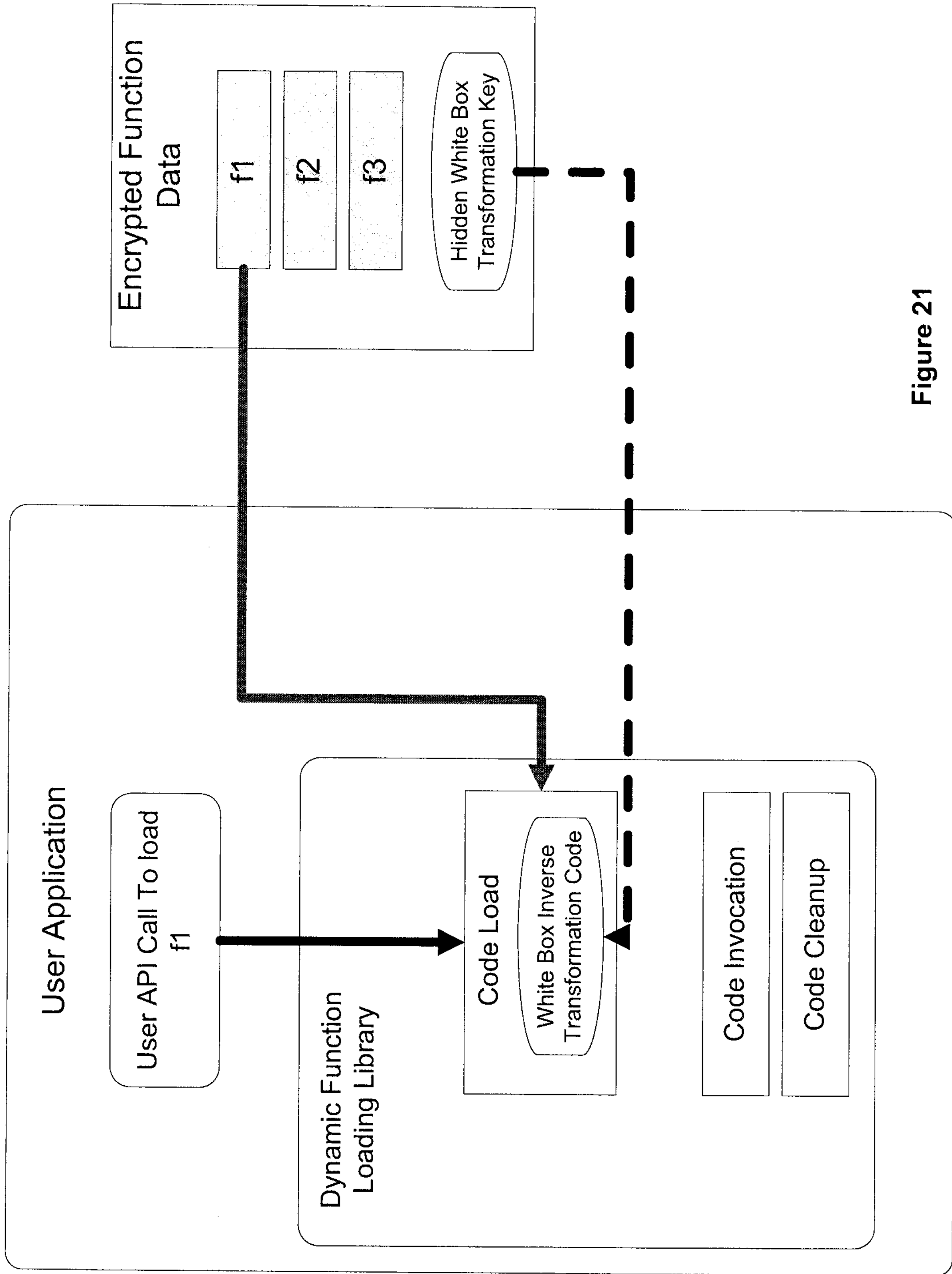


Figure 21

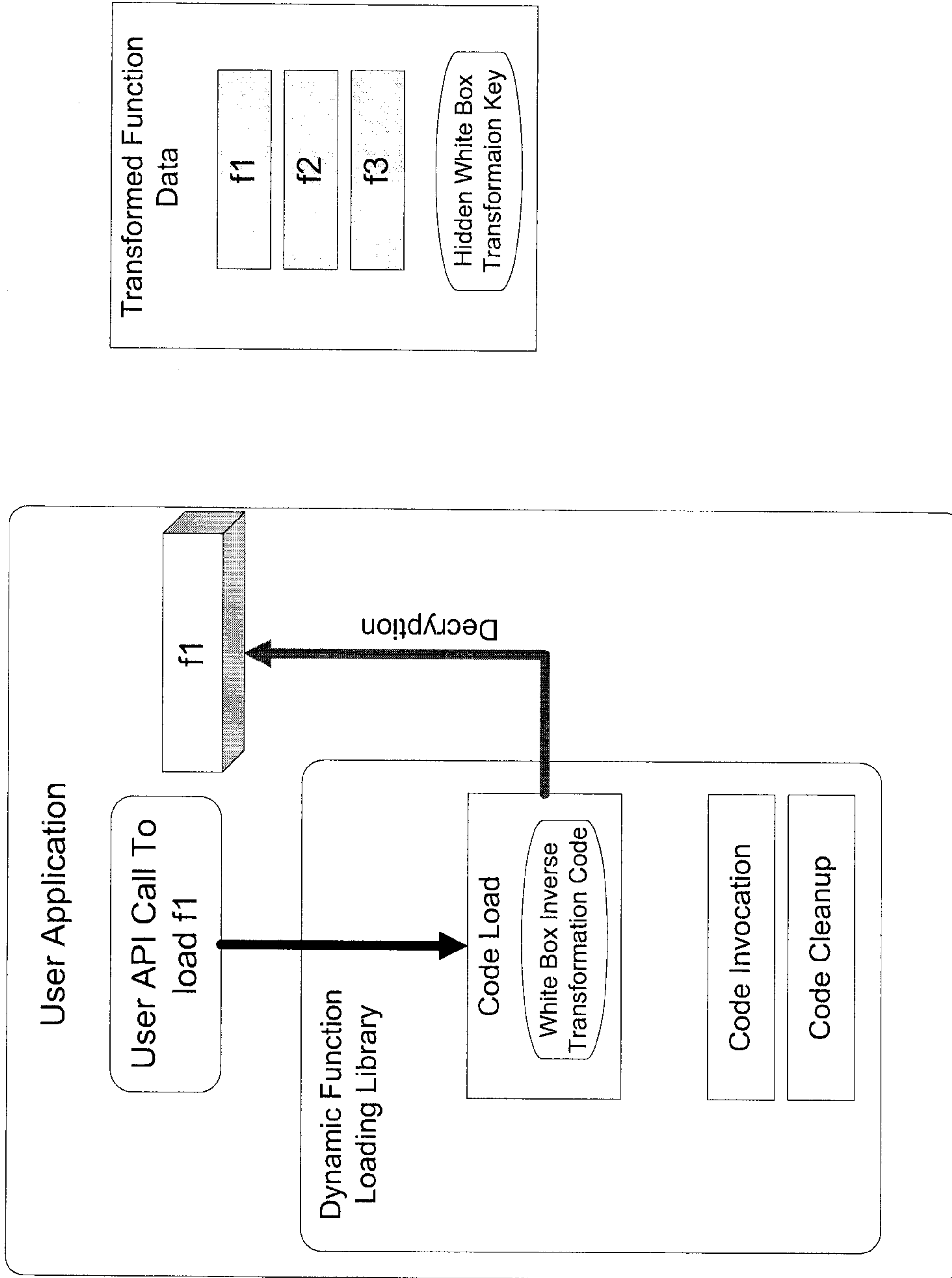


Figure 22

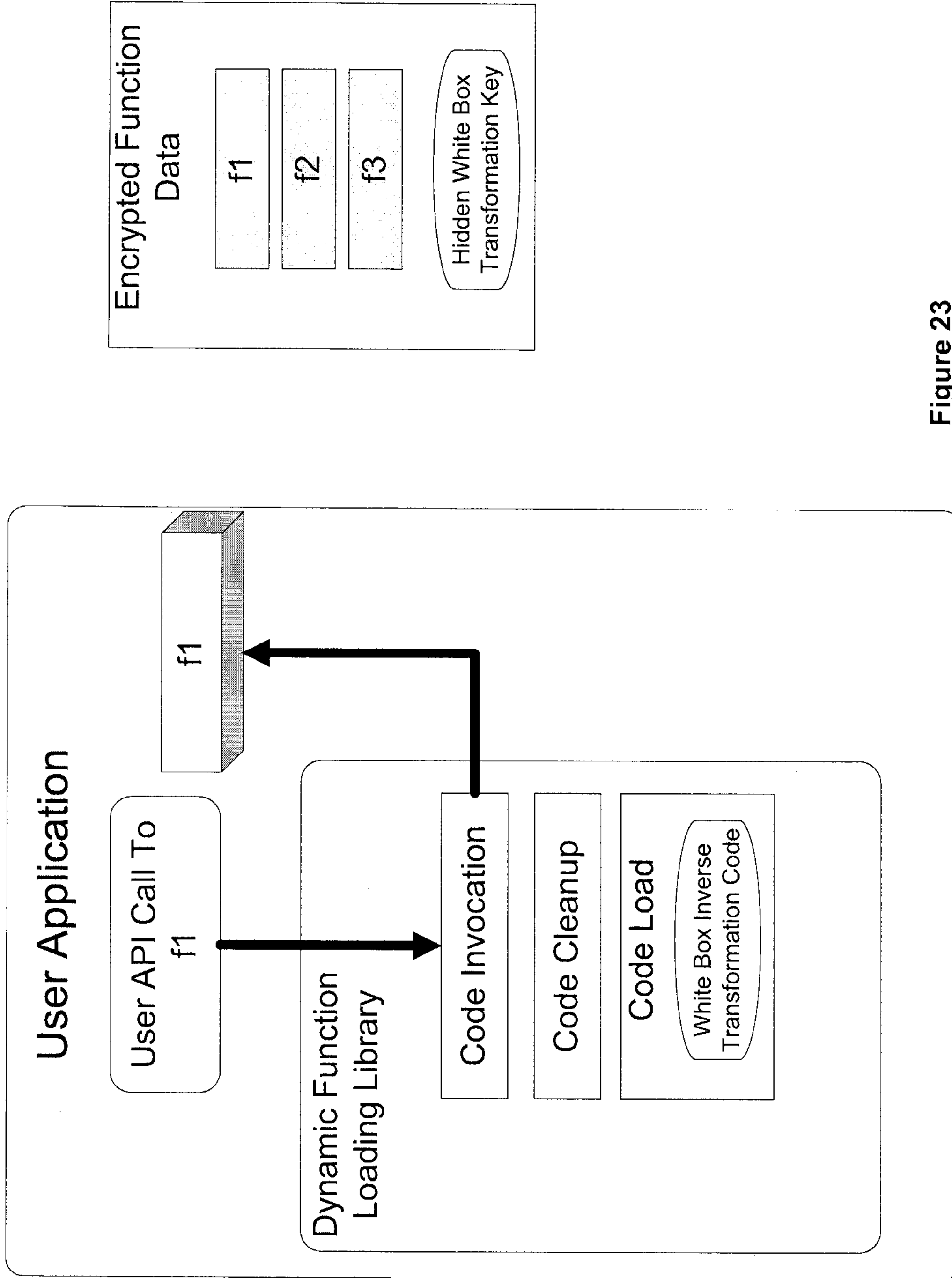


Figure 23

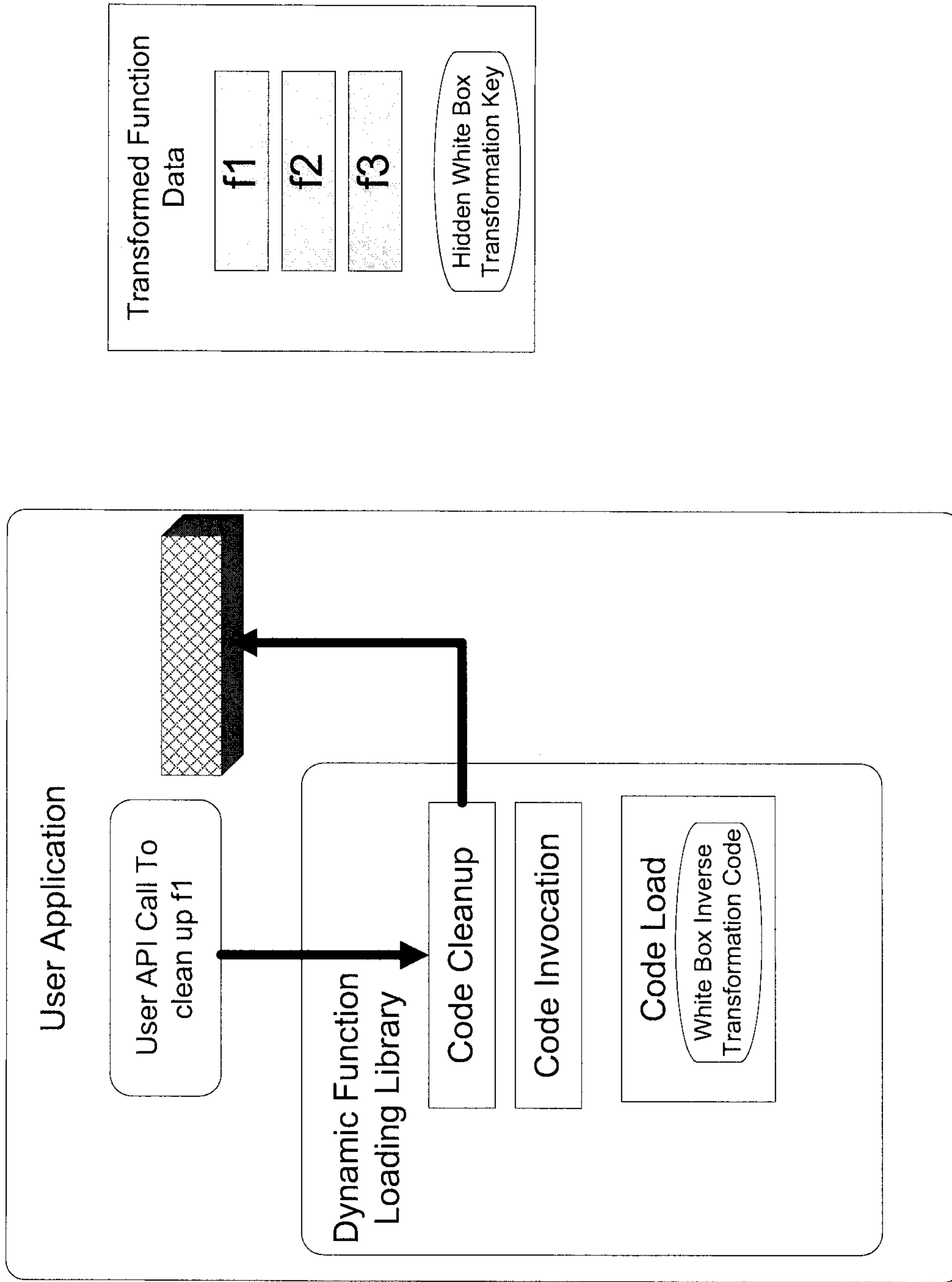


Figure 24

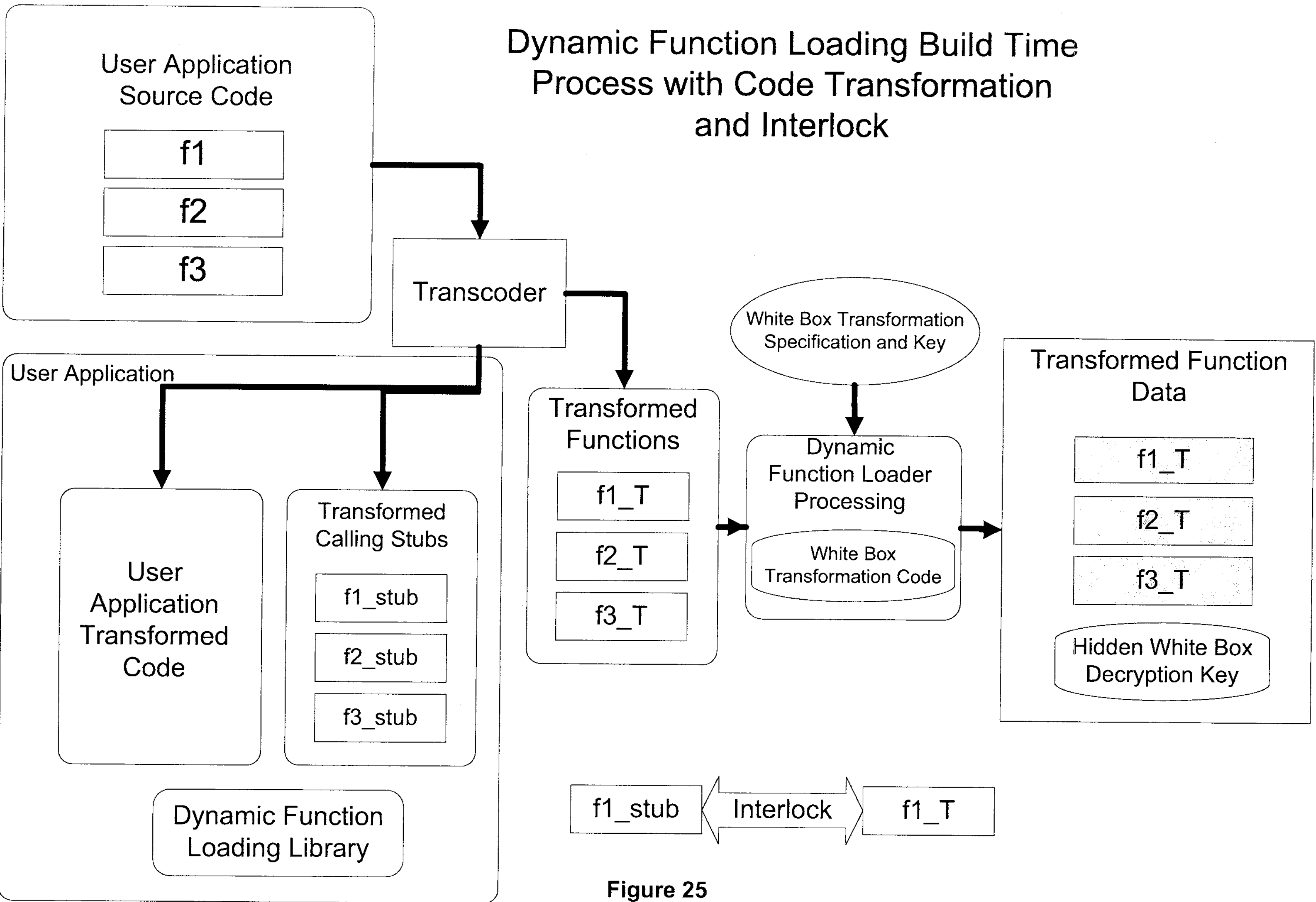


Figure 25

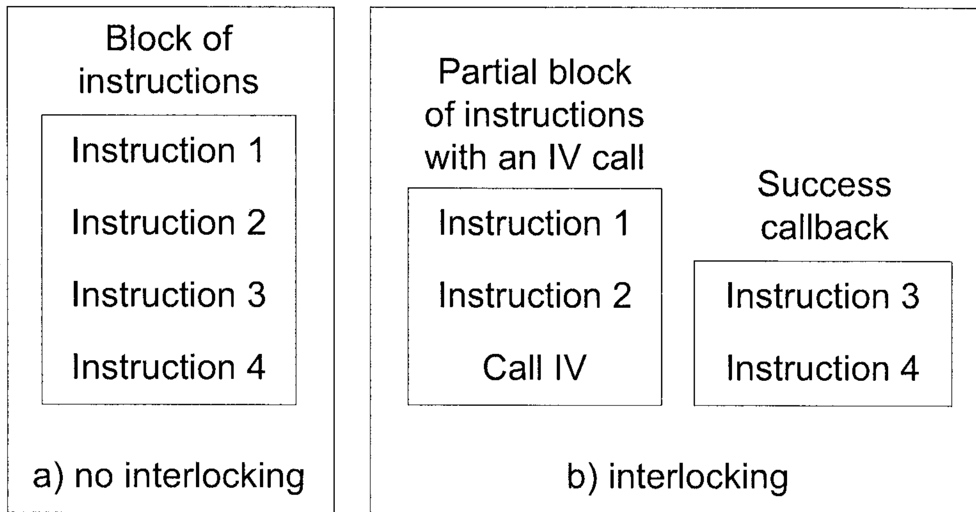


Figure 26

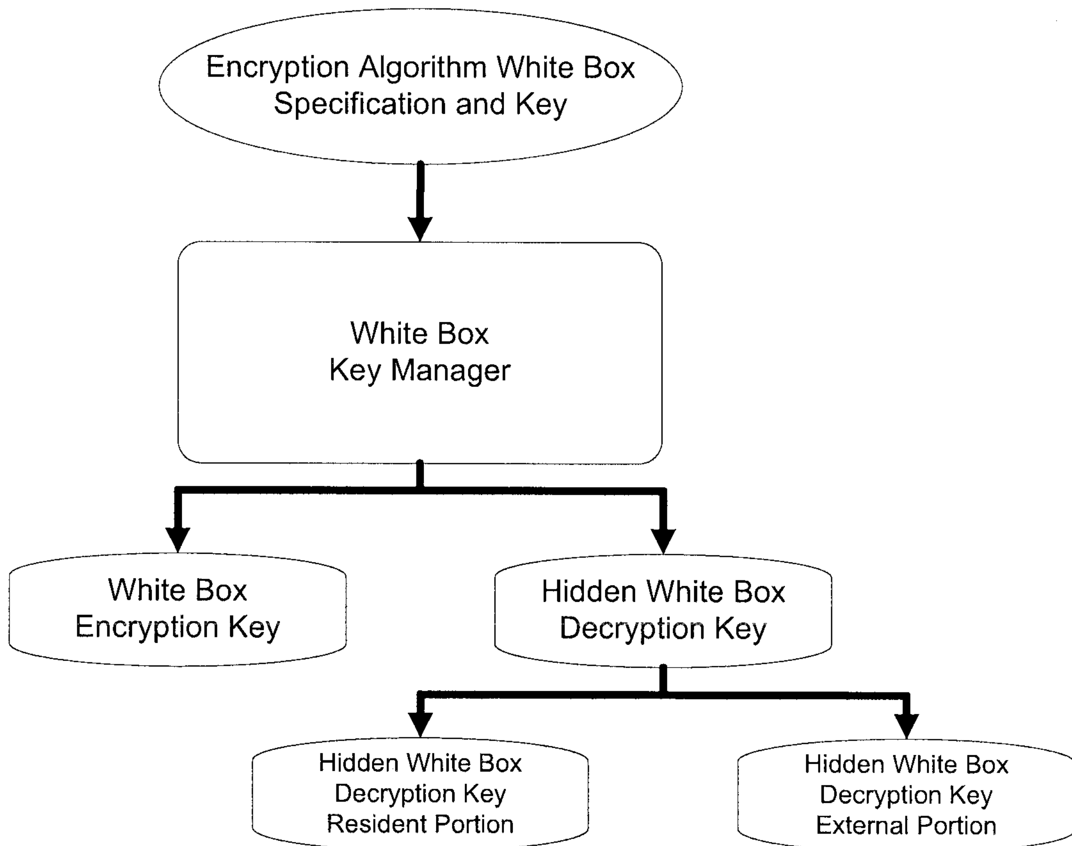


Figure 27

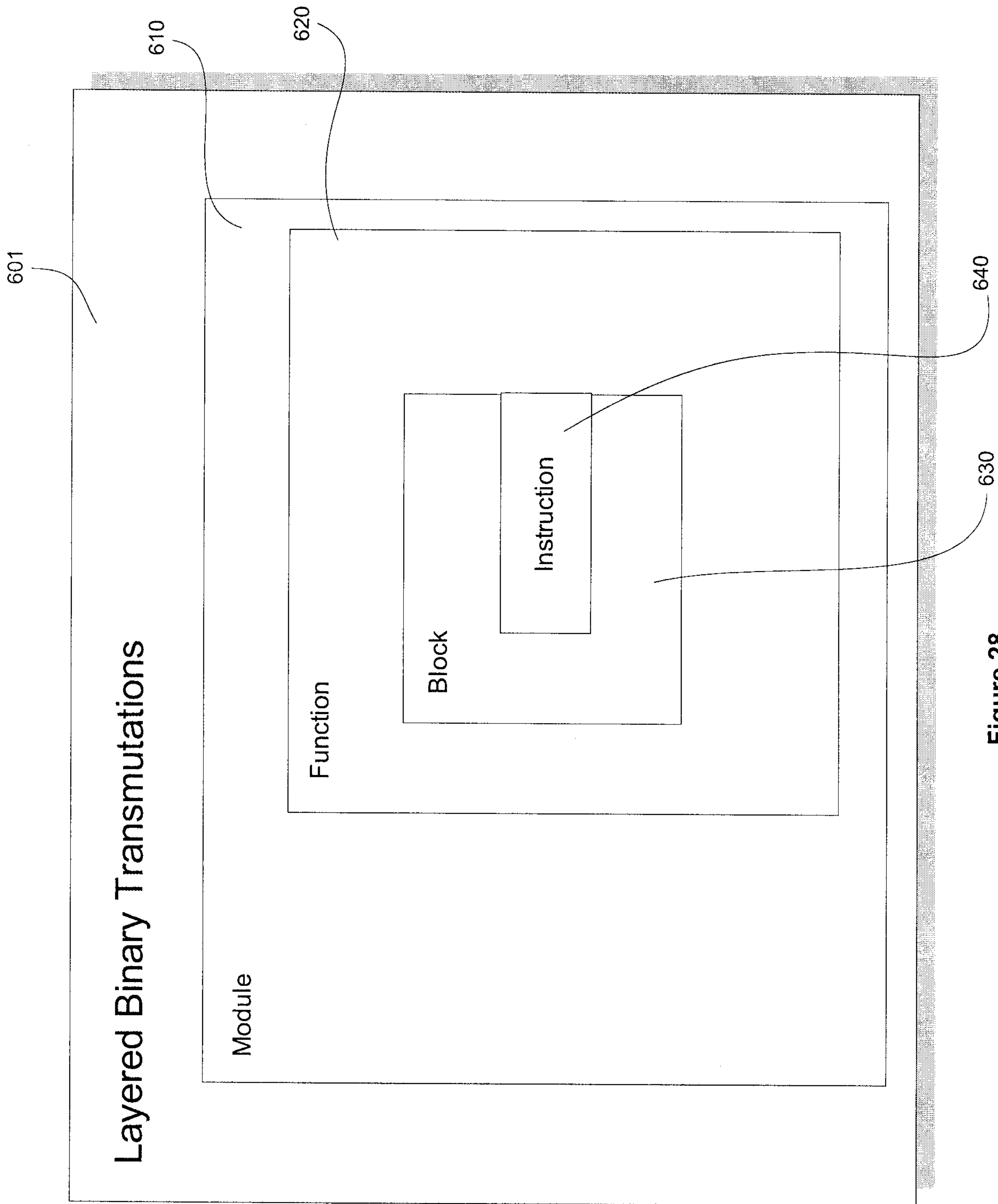


Figure 28

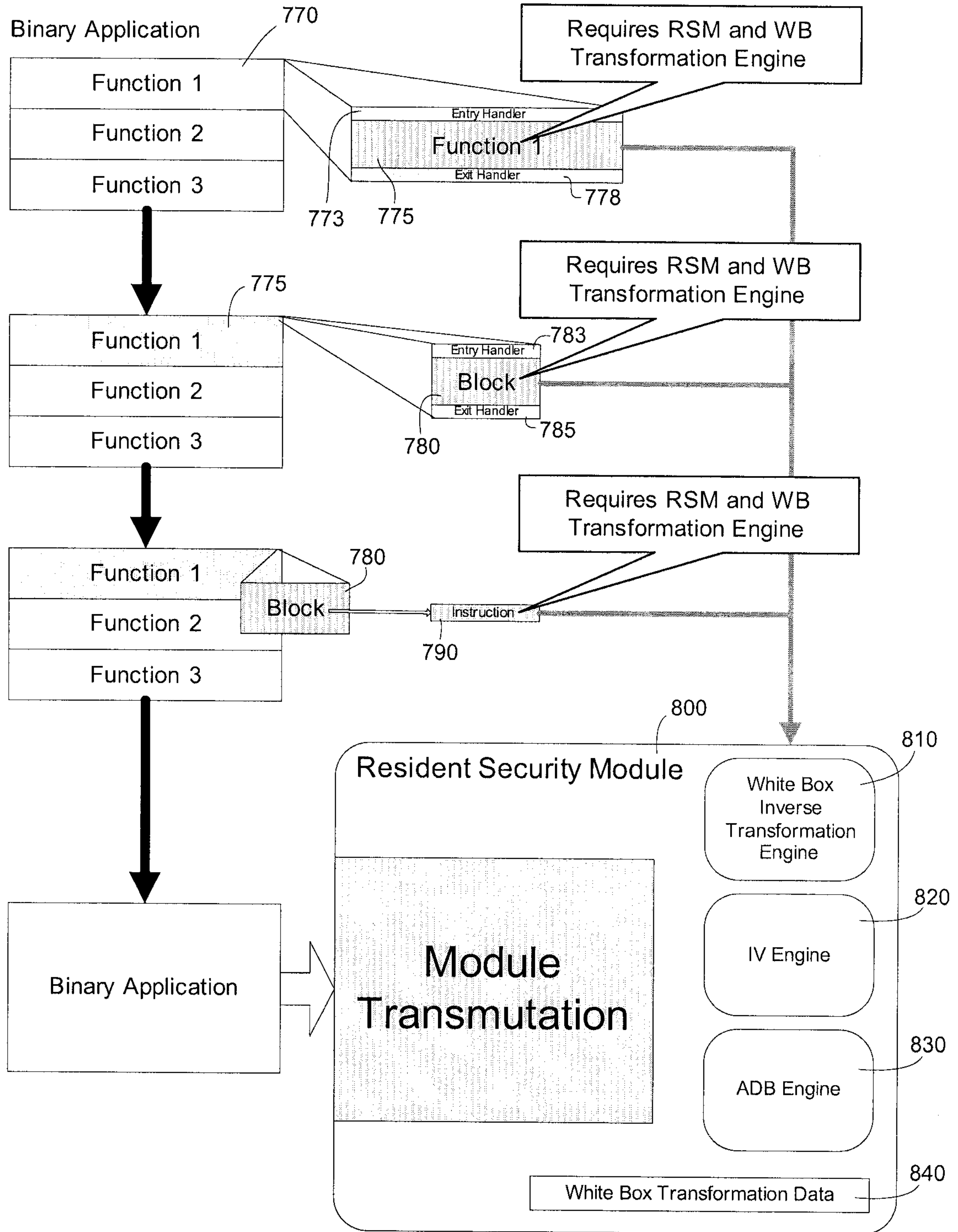


Figure 29

INTERNATIONAL SEARCH REPORT

International application No.
PCT/CA2010/000666

A. CLASSIFICATION OF SUBJECT MATTER
IPC: ***G06F 21/22*** (2006.01) , ***G06F 5/00*** (2006.01) , ***G06F 9/44*** (2006.01)
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F (2006.01)

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic database(s) consulted during the international search (name of database(s) and, where practicable, search terms used)
Databases: EPOQUE (EPODOC), Canadian Patents Database, Google Patents, IEEE XPLORE
Keywords: software, code, obfuscat*, tamper*, protect*, encrypt*, white*box, interlock*

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X Y	US 2006/0195703 A1 (Jakubowski) 31 August 2006 (31-08-2006) * paragraphs [0003], [0016], [0040] - [0042]; Fig 1 *	1, 2, 4 - 10, 37, 49, 50, 55 and 58 3, 38, 39, 40, 51 and 52
Y	US 2006/0031686 A1 (Atallah et al.) 09 February 2006 (09-02-2006) * Abstract; paragraphs [0057], [0151] and [0161] *	3, 38, 39, 40, 51 and 52
X	US 7,054,443 B1 (Jakubowski et al.) 30 May 2006 (30-05-2006) * Abstract; Fig 3 - 4; col 3, line 28 - col 8, line 33; col 11, lines 36 - 55 *	60 - 61

[ ] Further documents are listed in the continuation of Box C. [X] See patent family annex.

* Special categories of cited documents :
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search
2 August 2010 (02-08-2010)

Date of mailing of the international search report
9 August 2010 (09-08-2010)

Name and mailing address of the ISA/CA
Canadian Intellectual Property Office
Place du Portage I, C114 - 1st Floor, Box PCT
50 Victoria Street
Gatineau, Quebec K1A 0C9
Facsimile No.: 001-819-953-2476

Authorized officer
Raghid Shreih (819) 994-2694

INTERNATIONAL SEARCH REPORT
Information on patent family members

International application No.
PCT/CA2010/000666

Patent Document Cited in Search Report	Publication Date	Patent Family Member(s)	Publication Date
US2006195703A1	31 August 2006 (31-08-2006)	US2006195703A1 US7587616B2	31 August 2006 (31-08-2006) 08 September 2009 (08-09-2009)
US2006031686A1	09 February 2006 (09-02-2006)	EP1368927A1 US6941463B1 US2003018608A1 US6957341B2 US7287166B1 US2006107070A1 US7707433B2 US7757097B2 WO02076014A1 WO02076014A8	10 December 2003 (10-12-2003) 06 September 2005 (06-09-2005) 23 January 2003 (23-01-2003) 18 October 2005 (18-10-2005) 23 October 2007 (23-10-2007) 18 May 2006 (18-05-2006) 27 April 2010 (27-04-2010) 13 July 2010 (13-07-2010) 26 September 2002 (26-09-2002) 03 January 2003 (03-01-2003)
US7054443B1	30 May 2006 (30-05-2006)	US7054443B1 US7080257B1 US7277541B1 US2006136750A1 US7447912B2	30 May 2006 (30-05-2006) 18 July 2006 (18-07-2006) 02 October 2007 (02-10-2007) 22 June 2006 (22-06-2006) 04 November 2008 (04-11-2008)