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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
29 June 2006 (29.06.2006)

PCT

(10) International Publication Number
WO 2006/069335 A2

(51) International Patent Classification:
G06F 9/445 (2006.01)

(21) International Application Number:
PCT/US2005/046860

(22) International Filing Date:
21 December 2005 (21.12.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/638,298	21 December 2004 (21.12.2004)	US
11/316,621	19 December 2005 (19.12.2005)	US

(71) Applicant *(for all designated States except US)*: **NTT DOCOMO, INC.** [JP/JP]; Sanno Park Tower, 11-1, Nagata-cho, 2-chome, Chiyoda-ku, Tokyo 100-6150 (JP).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **YU, Dachuan** [CN/US]; Apartment 905, 3770 Flora Vista Avenue, Santa Clara, CA 95051 (US). **ISLAM, Nayeem** [US/US]; 3370 Coak Oak Way, Palo Alto, CA 94303 (US).

(74) Agents: **MALLIE, Michael, J.** et al.; Blakely, Sokoloff, Taylor & Zafman LLP, 7th Floor, 12400 Wilshire Boulevard, Los Angeles, CA 90025 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) **Title:** INFORMATION FLOW ENFORCEMENT FOR RISC-STYLE ASSEMBLY CODE

(57) **Abstract:** A method, article of manufacture and apparatus for performing information flow enforcement are disclosed. In one embodiment, the method comprises receiving securely typed native code and performing verification with respect to information flow for the securely typed native code based on a security policy.

WO 2006/069335 A2

INFORMATION FLOW ENFORCEMENT FOR RISC-STYLE ASSEMBLY CODE

PRIORITY

[0001] The present patent application claims priority to and incorporated by reference the corresponding provisional patent application serial no. 60/638,298, titled, "Information Flow Enforcement for RISC-Style Assembly Code", filed on December 21, 2004.

FIELD OF THE INVENTION

[0002] The present invention is related to the field of program execution and security; more specifically, the present invention is related to enforcing information flow constraints on assembly code.

BACKGROUND OF THE INVENTION

[0003] It is well-known that traditional security mechanisms are insufficient in enforcing information flow policies. In recent years, much effort has been put on protecting the confidentiality of sensitive data using techniques based on programming language theory and implementation. These techniques analyze the flow of information inside a target system, and have the potential to overcome the drawbacks of many traditional security mechanisms. Unfortunately, the vast amount of language-based research on information flow still does not address the problem for assembly code or machine executables directly. The challenge there largely lies in working with the lack of high-level abstractions (e.g., program structures and data structures) and managing the extreme flexibility offered by assembly code (e.g., memory aliasing and first-class code pointers).

[0004] Nonetheless, it is desirable to enforce noninterference directly at a low-level. On the one hand, any high-level programs must be compiled into low-level code before they can be executed on a real machine. Compilation or optimization bugs may invalidate the security guarantee established for the source program, and potentially be exploited by a malicious party. On the other hand,

some applications are distributed (e.g., bytecode or native code for mobile computation) or even directly written (e.g., embedded systems, core system libraries) in assembly code. Hence enforcement at a low-level is sometimes a must.

[0005] With the growing reliance on networked information systems, the protection of confidential data becomes increasingly important. The problem is especially subtle for a computing system that both manipulates sensitive data and requires access to public information channels. Simple policies that restrict the access to either the sensitive data or the public channels (or a combination thereof) often prove too restrictive. A more desirable policy is that no information about the sensitive data can be inferred from observing the public channels, even though a computing system is granted access to both. Such a regulation of the flow of information is often referred to as information flow, and the policy that sensitive data should not affect public data is often called noninterference.

[0006] Whereas it is relatively easy to detect and prevent naive violations that directly give out sensitive data, it is much more difficult to prevent an application from sending out information that is sophisticatedly encoded. Traditional security mechanisms such as access control, firewalls, encryption and anti-virus fall short on enforcing the noninterference policy. On the one hand, noninterference posts seemingly conflicting requirements for conventional mechanisms: it allows the use of sensitive information, but restricts the flow of it. On the other hand, the violation of noninterference cannot be observed from monitoring a single execution of the program, yet such execution monitoring serves as the basis of many conventional mechanisms.

[0007] The problem of information flow can be abstracted as a program that operates on data of different security levels, *e.g.*, *low* and *high*. Low data (representing low security) are public data that can be observed by all principles; high data (representing high security) are secret data whose access is restricted. An information flow policy requires that no information about the high (secret) input can be inferred from observing the low (public) output. In general, the security levels can be generalized to a lattice.

[0008] Such an information flow policy concerns tracking the flow of information inside a target system. Although it is easy to detect explicit flows (e.g., through an assignment from a secret `h` to a public `l` with `l=h`), it is much harder to detect various forms of implicit flow. For example, the statement l=0; if h then `l`=1 involves an implicit flow of information from h to `l`. At runtime, if the `then` branch is not taken, a conventional security mechanism based on execution monitoring will not detect any violation. However, information about `h` can indeed be inferred from the result of `l`, because the fact that `l` remains 0 indicates that the value of `h` must also be 0.

[0009] Instead of observing a single execution, language-based techniques derive an assurance about the program's behavior by examining, and possibly instrumenting, the program code. In the above example, the information essentially leaks through the program counter (referred to herein as *pc*)—the fact that a branch is taken reflects information about the guard of the conditional. In response, a security type system typically tags the program counter with a security label. If the guard of a conditional concerns high data, then the branches are verified under a program counter with a high security label. Furthermore, no assignment to a low variable is allowed under a high program counter, preventing the above form of implicit flow.

Traditional mechanisms

[0010] Many traditional security mechanisms are based on execution monitoring (EM). Some representative examples include security kernels, reference monitors, access control and firewalls. These mechanisms enforce security by monitoring the execution of a target system, looking for potential violations to a security policy. Unfortunately, such EM can only enforce "safety properties". An information flow policy is not a "property" (whether an execution satisfies a policy depends on other possible executions), and hence cannot be enforced by EM.

[0011] Cryptographic protocols depend on unproven complexity-theoretic assumptions. Some of these assumptions have been shown to be false (*e.g.*, DES, SHA0, MD5). Commercial use of strong cryptography is also entangled in political and legal complications. Perhaps more importantly, cryptography only ensures the security of the communication channel, establishing that the code comes from a certain source. It alone cannot establish the safety of the application.

[0012] Anti-virus is another widely applied approach. Its limitation is well-known, namely, it is always one step behind the virus, because it is based on detecting certain patterns in the virus code.

Mandatory access control

[0013] Mandatory access control is a runtime enforcement mechanism developed by Fenton and Bell and LaPadula, and prescribed by the "orange book" of the US Department of Defense for secure systems. In this approach, simple confidentiality policies are encoded using security labels. Data items and the program execution are tagged with these labels. The flow of information is controlled based on these labels, which are manipulated and computed at runtime.

[0014] An obvious weakness of mandatory access control is that it incurs computational and storage overhead to calculate and store security labels. Perhaps more importantly, the enforcement is based on observing the runtime execution of the program. As discussed above, such runtime enforcement cannot effectively detect implicit flows that concern all possible execution paths of the program.

[0015] To obtain confidentiality in the presence of implicit flows, a process of using sensitivity labels is introduced. If the execution of the program may split into different paths based on confidential data, the process sensitivity labels is increased. This effect of monotonically increasing labels is known as label creep. It makes mandatory access control too restrictive to be generally useful, because the result of the label computation tend to be too sensitive for the intended use of the data.

Language-based approaches

[0016] Even though there has been much work that applies language-based techniques to information flow, most of them focused on high-level languages. Many high-level abstractions have been formally studied, including functions, exceptions, objects, and concurrency, and practical implementations have been carried out. Nonetheless, enforcing information flow at only a high level puts the compiler into the trusted computing base (TCB). Furthermore, the verification of software distributed or written in low-level code cannot be overlooked.

[0017] Barthe et al., in Security types preserving compilation, *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 2937 of LNCS, pages 2–15. Springer-Verlag, Jan. 2004, presents a security-type system for a bytecode language and a translation that preserves security types. This reference discloses a stack-based language. More importantly, their verification circumvents a main difficulty—the lack of program structures at a low-level—by introducing a trusted component that computes the dependence regions and postdominators for conditionals. This component is inside the TCB and must be trusted.

[0018] Avvenuti et al., in Java bytecode verification for secure information flow, *ACM SIGPLAN Notices*, 38(12):20-27, Dec. 2003, applied abstract interpretation to enforce information flow for a stack-based bytecode language. Besides the difference in the machine models, their work also relied on the computation of control flow graphs and postdominators.

[0019] Zdancewic and Myers, in Secure information flow via linear continuations, *Higher-Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002, use linear continuations to enforce noninterference at a low-level. Their language is based on variables and still much different from assembly language. In particular, linear continuations, although useful in enforcing a stack discipline that helps information flow analysis, is absent from conventional assembly code. Hence, further (trusted) compilation to native code is required.

[0020] Traditionally, certifying compilation is mostly carried out for standard type safety properties (*e.g.*, TAL, PCC, ECC). Certifying compilation has been applied to security policies. However, such systems is based on security automata, hence cannot enforce noninterference. Besides the work on security-type preserving compilation by Barthe *et al.* as discussed above, related issues for π-calculus with security types have also been studied. There remains no related solution proposed targeting RISC-style assembly code.

SUMMARY OF THE INVENTION

[0021] A method, article of manufacture and apparatus for performing information flow enforcement are disclosed. In one embodiment, the method comprises receiving securely typed native code and performing verification with respect to information flow for the securely typed native code based on a security policy.

BRIEF DESCRIPTION OF THE DRAWINGS

[0022] The present invention will be understood more fully from the detailed description given below and from the accompanying drawings of various embodiments of the invention, which, however, should not be taken to limit the invention to the specific embodiments, but are for explanation and understanding only.

[0023] **Figure 1A** is a flow diagram of a process for information flow enforcement.

[0024] **Figure 1B** illustrates an environment in which the information flow enforcement of Figure 1A may be implemented.

[0025] **Figure 2** illustrates a simple security system at a source language level.

[0026] **Figure 3** is a flow diagram of some program structures.

[0027] **Figure 4** illustrates an example of information flow through aliasing.

[0028] **Figure 5** illustrates example information flow through code pointer.

[0029] **Figure 6** illustrates example context coercion without branching.

[0030] **Figure 7** illustrates the benefit of low-level verification.

[0031] **Figure 8** is a flow diagram of managing security levels.

[0032] **Figure 9** is a flow diagram of establishing noninterference.

[0033] **Figure 10** is a flow diagram of verification of a program.

[0034] **Figure 11** is a flow diagram of verification of an instruction sequence.

[0035] **Figure 12** illustrates syntax of TAL_C.

[0036] **Figure 13** illustrates operations semantics of TAL_C.

[0037] **Figure 14** illustrates TAL_C typing judgments.

[0038] **Figure 15** illustrates TAL_C typing rules of non-instructions.

[0039] **Figure 16** illustrates typing rules of TAL_C instructions.

[0040] **Figure 17A** illustrates expression translation (part of certifying compilation)

[0041] **Figure 17B** illustrates program and procedure declaration translation (part of certifying compilation).

[0042] **Figure 17C** illustrates command translation (part of certifying compilation).

[0043] **Figure 18** illustrates an example of a security-polymorphic function.

[0044] **Figure 19** is a block diagram of one embodiment of a mobile device.

[0045] **Figure 20** is a block diagram of one embodiment of a computer system.

DETAILED DESCRIPTION OF THE PRESENT INVENTION

[0046] A type system for low-level information flow analysis is disclosed. In one embodiment, the system is compatible with Typed Assembly Language, and models key features of RISC code including memory tuples and first-class code pointers. A noninterference theorem articulates that well-typed programs respect confidentiality. A security-type preserving translation that targets the

system is also presented, as well as its soundness theorem. This illustrates the application of certifying compilation for noninterference. These language-based techniques are promising for protecting the confidentiality of sensitive data. For RISC style assembly code, such low-level verification is desirable because it yields a small trusted computing based. Furthermore, many applications are directly distributed in native code.

[0047] Embodiments of the present invention focus on RISC-style assembly code. In one embodiment, typing annotations are used to recover information about high-level program structures, and do not require extra trusted components for computing postdominators. Furthermore, the techniques set forth herein do not rely on extra constructs such as linear continuations or continuation stacks. An erasure semantics reduces programs in our language to normal assembly code.

[0048] As set forth below, a language-based approach is used in which the enforcement is based on analyzing the program code statically. It does not require computation and storage of security labels at runtime. Furthermore, inspecting the program code and annotations allows the detection of implicit flows without falling into the label creep.

[0049] Embodiments of the present invention addresses information flow enforcement at the assembly level. To the authors' knowledge, it is the first that enforces confidentiality directly for RISC-style assembly code.

[0050] In one embodiment, a Confidentially Typed Assembly Language (TAL_C) is used for information flow analysis and its proof of noninterference. In one embodiment, the system is designed to be compatible with Typed Assembly Language (TAL). It thus approaches a unified framework for security and conventional type safety.

[0051] In one embodiment, the system models key features of an assembly language, including heap and register file, memory tuples (aliasing), and first-class code pointers (higher-order functions). In this document, we discuss a formal result with a core language supporting the above features for ease of

understanding, but also informally discuss extensions such as, for example, polymorphic and existential types.

[0052] Although it is desirable to directly verify at an assembly level, it is more practical to develop programs in high-level languages. In one embodiment, a formal translation is presented from a security-typed imperative source language to TAL_C is performed. This illustrates the application of certifying compilation for noninterference. A type-preservation theorem is presented for the translation.

[0053] In the following description, numerous details are set forth to provide a more thorough explanation of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form, rather than in detail, in order to avoid obscuring the present invention.

[0054] Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0055] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and

processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0056] The present invention also relates to apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus.

[0057] The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose systems may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these systems will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

[0058] A machine-readable medium includes any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read only memory ("ROM"); random access memory ("RAM"); magnetic disk storage media; optical storage media; flash memory devices; electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.); etc.

Challenges of Assembly Code

[0059] There are a number of challenges in enforcing information flow for assembly code. First, high-level languages make use of virtually infinite number of variables, each of which can be assigned a fixed security label. In assembly code, the use of memory cells is similar. However, a finite number of registers are reused for different source level variables, as long as the liveness regions of the variables do not overlap. As a result, one cannot assign a fixed security label to a register.

[0060] Second, the control flow of an assembly program is not as structured. The body of a conditional is often not obvious, and generally indeterminable, from the program code. Hence the idea of using a security context to prevent implicit flow through conditionals cannot be easily carried out.

[0061] Third, assembly languages are very expressive. Aliasing between memory cells can be difficult to understand. The support for first-class code pointers (i.e., the reflection of higher-order functions at an assembly level) is very subtle. A code pointer may direct a program to different execution paths, even though no branching instruction is immediately present. Nonetheless, it is important to support these features, because even the compilation of a simple imperative language with only first-order procedures can require the use of higher-order functions—returning is typically implemented as an indirect jump through a return register.

[0062] Fourth, since it is not practical to always directly program in an assembly language, a low-level type system must be designed so that the typing annotations can be generated automatically, *e.g.*, through certifying compilation. The type system must be as least as expressive as a high-level type system, so that any well-typed source program can be translated into a well-typed assembly program.

[0063] Finally, it is desirable to include erasure semantics where type annotations have no effect at runtime. A security mechanism cannot be generally applied in practice if it incurs too much overhead. Similarly, it is also undesirable

to change the programming model for accommodating the verification needs. Such a model change indicates either a trusted compilation process or a different target machine.

Overview of Information Flow Enforcement for Assembly Code

[0064] Figure 1A is a flow diagram of a process for information flow enforcement. The process is performed by processing logic that may comprise hardware (e.g., circuitry, dedicated logic, etc.), software (such as is run on a general purpose computer system or a dedicated machine), or a combination of both.

[0065] Referring to Figure 1A, the process begins by processing logic receiving securely typed native code (processing block 101). In one embodiment, processing logic receives the code via downloading or retrieving the code from a network location.

[0066] In one embodiment, the securely typed native code comprises assembly code that has undergone a security-type preserving translation that includes annotating the assembly code with type information. The annotations may comprise operations to mark a beginning and an ending of a region of the code in which two execution paths based on predetermined information are encountered.

[0067] After receiving the code, processing logic performs verification with respect to information flow for the securely typed native code based on a security policy (processing block 102). Verification is performed on a device (e.g., a mobile device such as a cellular phone) prior to the device running the code. In one embodiment, processing logic performs verification by statically checking behavior of the code to determine whether the code does not violate the security policy. In one embodiment, the code does not violate the security (safety) policy if the code, when executed, would not cause information of an identified type to flow from a device executing the code. In other words, it verifies the information flow that would occur under control of the assembly code when executed.

[0068] If verification determines the code does not violate the security policy, processing logic removes any annotations from the code (processing block 103) and runs the code (processing logic 104).

[0069] Figure 1B illustrates an environment in which the information flow enforcement of Figure 1A may be implemented. Referring to Figure 1B, a program 150 is subjected to a security type inference 151 based on a security policy 152. The result is a securely typed program 153. A certifying compiler 154 compiles program 153 and, as a result, produces securely typed target code 155.

[0070] Securely typed target code 155 may be downloaded by a consumer device. The consumer device may be a cellular phone or other mobile device, such as, for example, described below. The consumer device runs a verification module 160 on securely typed target code 155 before running code 155. The verification module 160 performs the verification based on security policy 152, acting as a type checker.

[0071] The consumer device also runs an erasure module 170 on securely typed target code 155 to erase annotations that were added to the code by certifying compiler 154 before running code 155.

[0072] If the verification module 160 determines that the code is safe or otherwise verifies the code is acceptable based on security policy 152, verification module 160 signals the consumer device that securely typed target code 155 may be run by the consumer device (e.g., a processor on the consume device).

[0073] The following discussion describes in detail the information flow problem and the solution.

A High-Level Security Type System

[0074] Figure 2 shows an example of a two-level security-type system for a simple imperative language with first-order procedures. A program P comprises a list of procedure declarations F_i and a main command C. A procedure declaration documents the security level of the program counter with pc, indicating that the procedure body will only update variables with security levels no less than pc. A procedure also declares a list of arguments x_i under call-by-reference semantics. Commands C consist of assignments, sequential compositions, conditional

statements, while-loops, and procedure calls. Variables V cover both global variables v and procedure arguments x. Expressions E are formed by constants (i), variables, and their additions.

[0075] Referring to Figure 2, Rules [E1-4] relate expressions to security types (levels). Any expression may have type high (it is secure to treat any data as sensitive). Constants and low variables may have type low. An addition expression have type low if both sub-expressions have type low.

[0076] Rules [C1-7] track the security level of the program counter (pc) when verifying the commands. Assignments to high variables are always valid (Rule [C1]). However, an assignment to a low variable is valid only if both the expression and the pc are low (Rule [C2]). For a conditional (Rule [C3]), the security level of the sub-commands must match the security level of the guard expression; together with Rule [C2], this guarantees that low variables are not modified within a branch under a high guard. After a conditional, it is useful to reset the pc to low, avoiding a form of label creep, where monotonically increasing security labels are too restrictive to be generally useful. Such a context reset is achieved with a subsumption rule (Rule [C4]); intuitively, if it is secure to execute a command in a sensitive context, then it is also secure in an insensitive one. A sequential composition is verified so that both sub-commands are valid under the given pc (Rule [C5]). The handling of a while-loop is similar to that of a conditional statement (Rule [C6]). A procedure call is valid if pc matches the expected security level, and the arguments have the expected types (Rule [C7]); note that only variables (v or x) may server as the arguments, which are handled by reference (also know as ``in-out'' arguments)

[0077] Finally, a procedure declaration is valid if the body can be verified under the expected pc and arguments (Rule [F1]). A program is valid if all procedure declarations and the main command are valid (Rule [P1]).

Explicit Assignment

[0078] One way of transferring information in a high-level language is through assignment. As discussed above, variables in a high-level language can be "tagged" with security labels such as low and high. The security-type system prevents label mismatch for assignments. At an assembly level, memory cells can be tagged similarly. When storing into a memory cell, a typing rule ensures that the security label of the source matches that of the target.

[0079] Regulating information flow through registers is different, because registers can be reused for different variables with different security labels. Since variable and liveness information is not available at an assembly level, one cannot easily base the enforcement upon that.

[0080] In fact, a similar problem arises even for normal type safety. A register in Type Assembly Language (TAL) can have different types at different program points. These types are essentially inferred from the computation itself. For instance, in an addition instruction `add` r_d, r_s, r_t, the register r_d is given the type `int`, because only int can be valid here. Similarly, when loading from a memory cell, the target register is given the type of the source memory cell. We adapt such inference for security labels. In the addition `add` r_d, r_s, r_t, the label of r_d is obtained by joining the labels of r_s and r_t, because the result in r_d reflects information from both r_s and r_t. Moving and memory reading instructions are handled similarly.

Program Structure

[0081] A conditional statement in a high-level program can be verified so that both subcommands respect the security level of the guard expression. Such verification becomes difficult in assembly code, where the "flattened" control flow provides little help in identifying the program structure. A conditional is typically translated into a branching instruction (`bnz` r, l) and some code blocks, where the postdominator of the two branches are no longer apparent.

[0082] In one embodiment, annotations are used to restore the program structure by pointing out the postdominators whenever they are needed. Note that

high-level programs provide sufficient information for deciding the postdominators, and these postdominators can always be statically determined. For instance, the end of a conditional command is the postdominator of the two branches. Hence, a compiler can generate the annotations automatically based on a securely typed source program. In one embodiment of the system of the present invention, the postdominator annotation is a static code label paired with a security label.

[0083] Since branching instructions (`bnz` *r*, *l*) are the only instructions that could directly result in different execution paths, it would appear that one should enhance branching instructions with postdomonators. The typing rule then checks both branches under a proper security context that takes into account the guard expression. Such a security context terminates when the postdominator is reached.

[0084] Although plausible, this approach is awkward. Figure 3 demonstrates three scenarios. Besides the conditional scenario, branching instructions are also used to implement while-loops, where the postdominator is exactly the beginning of one of the branches. In this case, only the other branch should be checked under a new security context. If the branching instruction is directly annotated, the corresponding typing rule would be "overloaded." More importantly, an assembly program may contain "implicit branches" where no branching instruction is present. The third scenario illustrates that an indirect jump may lead the program to different paths based on the value of its operand register. A concrete example will appear below.

[0085] Inspiration of a better solution lies in the simple system of Figure 2. Note that the subsumption rule [C4] is not tied to any particular commands. It essentially marks a region of computation where the security level is raised from low to high. The end of the region is exactly a postdominator. Following this, in one embodiment, the approach set forth herein mimics the high-level subsumption rule with two low-level *raising* and *lowering* operations that explicitly manipulate the security context and mark the beginning and end of the secured region.

Memory Aliasing

[0086] Aliasing of memory cells present another channel for information transfer. In Figure 4, a low pointer p_l and a high pointer p_h are aliases of the same cell (they are two pointers pointing to the same value). The code in the same figure may change the aliasing relation based on some high variable `h` by letting p_h point to another cell. Further modification through p_h may or may not change the value stored in the original cell. As a result, observing through the low pointer p_l gives out information about the high variable `h`.

[0087] The problem lies in the assignment through the high pointer p_h, because it reveals information about the aliasing relation. In one embodiment, pointers are tagged with two security labels. One is for the pointer itself, and the other is for the data being referenced. In one embodiment, assignments to low data through high pointers are not allowed. This is a conservative approach—all pointers are considered as potential aliases.

Code Pointers

[0088] Code pointers further complicate information flow. Figure 5 shows a piece of functional code where `f` represents different functions based on a high variable `h`. In its reflection at an assembly level, different code labels will be assigned to `f` based on the value of `h`. Naturally, `f` contains sensitive information and should be labeled high. However, the actual functions `f0` and `f1` can only be executed under a low context, because they modify a low variable `l`. In this case, the invocation to `f` should be prohibited.

[0089] In one embodiment of the system of the present invention, similar to data pointers, code pointers are also given two security labels. The typing rules ensure that no low function is called through a high code pointer.

Security Context Coercion

[0090] Figure 6 shows a piece of code where a mutable code pointer complicates the flow analysis. Functions `f0` and `f1` only modify high data. A

reference cell `f` is assigned different code pointers within a high conditional. Later, the reference cell `f` is dereferenced and invoked in a low context.

[0091] This code is safe with respect to information flow. At a high level, a subsumption rule like Rule [C4] in Figure 2 allows calling the high function `!f( )` in a low context. However, in its assembly counterparts, both the calling to `f` and the returning from `f` are implemented as indirect jumps. The calling sequence transfers the control from a low context to a high context, whereas the returning sequence does the opposite. Since the function invocation is no longer atomic at an assembly level, one cannot directly devise a subsumption rule. Furthermore, there is no explicit branching instruction present when `f` is dereferenced and invoked (the third scenario of Figure 3).

[0092] In one embodiment of the system of the present invention, the raising and lowering operations explicitly mark the boundary of the subsumption rule. During certifying compilation, the source-level typing and program structure provide sufficient information for generating the target-level annotations. When a subsumption rule is applied in the source code, the corresponding target code is generated within a pair of raising and lowering operations.

Enforcing Information Flow Policies

[0093] As discussed above, embodiments of the present invention enforce information flow policies directly for assembly code. A benefit of this approach is illustrated in Figure 7A and B. As shown in Figure 7A, existing language-based approaches enforce information flow using security-type system for high-level languages (*e.g.*, Java). Verification is achieved at the source level only. However, a high-level program must be compiled before executing on a real machine. A compiler performs most of the transformation, including generating the native code. Translation or optimization bugs may invalidate the security guarantee established for the source program. As a result, such source-level verification relies on a huge trusted computing base.

[0094] In contrast, a security-type system is set forth herein for verifying assembly code directly. As shown in Figure 7B, verification is achieved on

securely typed native code. This removes much of the compiler out of the trusted computing base, thereby achieving a trustworthy environment. Furthermore, this allows the security verification of programs directly distributed in native code.

[0095] An embodiment of the security-type system of the present invention relies on a security context to prevent implicit flows that result from the program structure. The security context is explicitly manipulated by two operations *raise* and *lower*. Figure 8 illustrates an example path. At the point where a program may branch into different execution paths based on sensitive data, the security context is raised high enough to capture the sensitivity of the data. In Figure 8, this occurs at point 801 and 802 in the program that runs from P_{START} to P_{end}. At the place where the different execution paths join together (*i.e.*, a postdominator), the security context is lowered to its original level. In Figure 8, this occurs at points 803 and 804 in the program that runs from P_{start} to P_{end}. Hence, the program code can be statically viewed as organized into different security regions, whose beginning and ending are explicitly marked by *raise* and *lower*.

[0096] Given a security level θ of concern, any data item can be viewed as either public or secret, based on the comparison between its security level and θ. The desired noninterference result is that public output data reflects no information about secret input data. In one embodiment, a noninterference result is established based on an equivalence relation $\approx_\theta$. Intuitively, two machine states are equivalent with respect to security level θ if they contain the same public data. Figure 9 shows two execution paths of the same program based on different, but equivalent, inputs. Under a low security context, the two executions match each other in a lock-step manner. Under a high security context, the two executions may involve different code. However, an embodiment of the system of the present invention makes sure that no low data is updated under a high security context. Thus, following the transitivity of the equivalence relation, the two executions join at the postdominator with equivalent states.

[0097] One embodiment of the system of the present invention provides an encoding of confidentiality information in type annotations. The verification

process is guided by some typing rules. Figure 10 is a flow diagram of one embodiment of a process for verifying a program against its type annotations. This process delegates the task to three components, verifying the heap, the register file and the instruction sequence respectively. The program is secure with respect to a security policy only if all the three components return successfully. The process is performed by processing logic that may comprise hardware (circuitry, dedicated logic, etc.), software (such as is run on a general purpose computer system or a dedicated machine), or a combination of both.

[0098] The verification of an instruction sequence is the most complex part. Nonetheless, it is fully syntactic, thereby allowing a straightforward and mechanical implementation. Based on the syntax of the current instruction, the verification is carried out against different typing rules. The verification aborts whenever a typing rule is not satisfied, reporting a violation of confidentiality. If the typing rule is satisfied on the current instruction, the verification proceeds recursively on the remainder instruction sequence. Finally, if the end of the instruction sequence is reached (i.e., `jmp` or `halt`), processing logic terminates the verification after checking the corresponding rules.

[0099] In one embodiment, the formal rules set forth in Figures 14, 15 and 16 are used, and explained below.

[00100] Referring to Figure 10, the process begins by processing logic tests whether H is verifiable against Ψ (processing block 1002). If it fails, processing logic indicates the program is not acceptable (processing block 1010). If it is, processing logic tests whether R is verifiable agent (Ψ, Γ) (processing block 1003). If it is not, processing logic indicates that the program is not acceptable (processing block 1010). If it is, processing logic tests whether S is verifiable against (Ψ, Γ, κ) (processing block 1004). If it is, processing logic indicates the program is acceptable (processing block 1011).

[00101] Figure 11 illustrates an example flow diagram for verification of an instruction sequence.

Abstract Machine

[00102] In one embodiment, language TAL_C resembles TAL and STAL for ease of integration with existing results on conventional type safety. Some additional constructs are used for confidentiality, while some TAL and STAL features that are orthogonal to the proposed security operations are removed. Security labels are assumed to form a lattice $\mathcal{L}$. The symbol θ is used to range over elements of $\mathcal{L}$. The symbols $\bot$ and $\top$ are used as the bottom and top of the lattice, $\cup$ and $\cap$ as the lattice join and meet operations, $\subseteq$ as the lattice ordering. The following explains the syntactic constructs of TAL_C.

[00103] The top portion of Figure 12 presents the type constructs. Security contexts are referred to as κ. An empty security context ($\bullet$) represents an program counter with the lowest security label. A concrete context ($\theta \triangleright w$) is made up of a security label θ (the current security level) and a postdominator w. The postdominator w has the syntax of a word value, but its use is restricted by the semantics to be eventually an instantiated code label, i.e., the ending point of the current security level. The postdominator w could also be a variable α; this is useful for compiling procedures, which can be called in different contexts with different postdominators.

[00104] Pre-types τ reflect the normal types as seen in TAL, including integer types, tuple types, and code types. In comparison with TAL, in one embodiment, the code type described herein requires an extra security context (κ) as part of the interface. A type (σ) is either a pre-type tagged with a security label or a nonsense type (*ns*) for uninitialized stack slots. A stack type (Σ) is either a variable (ρ), or a (possibly empty) sequence of types. The variable context (Δ) is used for typing polymorphic code; it documents stack type variables (ρ) and postdominator variables (α). Stack types and postdominators are also generally referred to herein as type arguments ψ. Finally, heap types (Ψ) or register file types (Γ) are mappings from heap labels or registers to types; the sp in the register file represents the stack.

[00105] The middle portion of Figure 12 shows the value constructs. A word value w is either a variable, a heap label l, an immediate integer i, a nonsense

value for an uninitialized stack slot, or another word value instantiated with a type argument. Small values *v* serve as the operands of some instructions; they are either registers *r*, word values *w*, or instantiated small values. Heap values h are either tuples or typed code sequences; they are the building blocks of the heap *H*. Note that a value does not carry a security label. This is consistent with the philosophy that a value is not intrinsically sensitive--it is sensitive only if it comes from a sensitive location, which is documented in the corresponding types (Ψ and Γ). Finally, a register file *R* stores the contents of all registers and the stack, where the stack is a (possibly empty) sequence of word values.

[00106] Code constructs are given in the bottom portion of Figure12. A minimal set of instructions from TAL and STAL is retained, and two new instructions (`raise` κ and `lower` *l*) are introduced for manipulating the security context as discussed above. In one embodiment, a program is the usual triple tagged with a security context. The security context facilitates the formal soundness proof, but does not affect the computation.

[00107] In the operational semantics (Figure 13), there are only two cases that modify the security context: `raise` κ' updates the security context to κ', and `lower` *w* picks up a new security context from the interface of the target code *w*. In all other cases, the security context remains the same, and the semantics is standard. The operational semantics mimics the behavior of a real machine. One can obtain a conventional machine by removing the security contexts and `raise` κ instructions, and replacing `lower` *w* with `jmp` *w*.

Typing Rules

[00108] The static semantics consists of judgment forms summarized in Figure 14. A security context appears in the judgment of a valid instruction sequence. Heap and register file types are made explicit in the judgment of a valid program for facilitating the noninterference theorem. All other judgment forms closely resemble those of TAL and STAL.

[00109] The typing rules are given in Figures 15 and 16. A type construct is valid (top six judgment forms in Figure 14 if all free type variables are

documented in the type environment. Heap values and integers may have any security label. The types of heap labels and registers are as described in the heap type and the register file type respectively. All other rules for non-instructions are straightforward.

[00110] In one embodiment, a macro $SL(\kappa)$ is used to refer to the security label component of κ. $SL(\bullet)$ is defined to be $\perp$. The typing rules for `add`, `ld` and `mov` instructions infer the security labels for the destination registers; they take into account the security labels of the source and target operands and the current security context.

[00111] The rule for `bnz` first checks that the guard register r is an integer and the target value v is a code label. It then checks that the current security context is high enough to cover the security levels of the guard (preventing flows through program structures) and the target code (preventing flows through code pointers). Lastly, the checks on the register file and the remainder instruction sequence make sure that both branches are secure to execute.

[00112] The rule for `st` concerns four security labels. This rule ensures that the label of the target cell is higher than or equal to those of the context, the containing tuple, and the source value.

[00113] The rules for the stack instructions follow similar ideas. In essence, the stack can be viewed as an infinite number of registers. Instruction `salloc` or `sfree` add new slots to or remove existing slots from the slot, so the rules check the remainder instruction sequence under an updated stack type. The rule for instruction `sld` or `sst` can be understood following that of the `mov` instruction.

[00114] The rule for `raise` checks that the new security context is higher than the current one. Moreover, it looks at the postdominator w' of the new context, and makes sure that the security context at w' matches the current one. The remainder instruction sequence is checked under the new context.

[00115] Since the rule for `raise` already checked the validity of the ending label of a secured region, the task for ending the region is relatively simple. The rule for `lower` checks that its operand label matches that dictated by the security context. This guarantees that a secured region be enclosed within a `raise`-

`lower` pair. The rule also makes sure that the code at w is safe to execute, which involves checking the security labels and the register file types.

[00116] The rule for `jmp` checks that the target code is safe to execute. Similar checks also appeared in the rule for `bnz`. In these two rules, the security context of the target code is the same as the current one. This is because context changes are separated from conventional instructions in one embodiment of the system. For example, one may enclose high target code within `raise` and `lower` before calling it in a low context.

[00117] Finally, halting is valid only if the security context is empty, and the value in r_1 has the expected type σ.

[00118] The TAL_C language enjoys conventional type safety (memory and control flow safety), which can be established following the progress and preservation lemmas. The proofs of these lemmas are similar to those of TAL and STAL and have been omitted to avoid obscuring the present invention.

Lemma 1 (Progress): If $\Psi; \Gamma \vdash P$ then either:

1. there exists P' such that $P \mapsto P'$, or
2. P is of the form $(H, R\{r_1 \mapsto w\}, \text{halt}[\sigma])$. where $\vdash H : \Psi$ and $\Psi;^{\circ} \vdash w : \sigma$.

Lemma 2 (Preservation): If $\Psi; \Gamma \vdash P$ and $P \mapsto P'$, then there exists Γ' such that $\Psi; \Gamma' \vdash P'$.

[00119] Before presenting the noninterference theorem for TAL_C, the equivalence of two programs is defined with respect to a given security level θ.

Definition 1 (Heap Equivalence): $\Psi \vdash H_1 \approx_\theta H_2 \Leftrightarrow$ for every $l \in \text{dom}(\Psi)$,

$\Psi(l) = \tau_{\theta'}$ and $\theta' \subseteq \theta$ implies $H_1(l) = H_2(l)$.

Definition 2 (Stack Equivalence): $\Sigma \vdash S_1 \approx_\theta S_2 \Leftrightarrow$ for every stack slot $i \in \text{dom}(\Sigma)$,

$\Sigma(i) = \tau_{\theta'}$ and $\theta' \subseteq \theta$ implies $S_1(i) = S_2(i)$.

Definition 3 (Register File Equivalence): $\Gamma \vdash R_1 \approx_\theta R_2 \Leftrightarrow$ both

1. $\Gamma(\text{sp}) \vdash R_1(\text{sp}) \approx_\theta R_2(\text{sp})$, and

2. for every $r \in \mathrm{dom}(\Gamma)$, $\Gamma(r) = \tau_{\theta'}$ and $\theta' \subseteq \theta$ implies $R_1(r) = R_2(r)$.

Definition 4 (Program Equivalence): $\Psi; \Gamma \vdash P_1 \approx_\theta P_2 \Leftrightarrow P_1 = (H_1, R_1, I_1)_{\kappa_1}$,

$P_2 = (H_2, R_2, I_2)_{\kappa 2}$, $\Psi \vdash H_1 \approx_\theta H_2$, $\Psi; \Gamma \vdash R_1 \approx_\theta R_2$, and either:

1. $\kappa_1 = \kappa_2$, $SL(\kappa_1) \subseteq \theta$, and $I_1 = I_2$, or

2. $SL(\kappa_1) \not\subseteq \theta, SL(\kappa 2) \not\subseteq \theta$.

[00120] The above three relations are all reflexive, symmetrical, and transitive. The noninterference theorem relates the executions of two equivalent programs that both start in a low security context (relative to the security level of concern). If both executions terminate, then the result programs must also be equivalent.

[00121] The basic idea of the proof is intuitive. Based on the security context of the programs and the security level of concern, the executions can be phased into "low steps" and "high steps." The two executions under a low step can be related, because they are executing the same instructions. Reasoning under a high step is different—the two executions are no longer in lock step. However, the raise and lower mark the beginning and end of a secured region, and therefore the program states are related before the `raise` and after the `lower`, hence circumvent directly relating two executions in a high step. Additional formal details with three lemmas and a noninterference theorem are provided. Lemma 3 indicates that a security context in a high step can be changed only with raise or lower. Lemma 4 states that a terminating program is to reduce to a step that discharges the current security context with a lower. Lemma 5 articulates the lock step relation between two equivalent programs in a low step. Theorem 1 then follows from these lemmas.

[00122] In the following, $\mapsto^*$ represents the reflexive and transitive closure of $\mapsto$. $\Sigma \succeq_\theta \Sigma'$ means that $\Sigma(i) = \Sigma'(i)$ for every i such that $\Sigma'(i) = \tau_{\theta'}$ and $\theta' \subseteq \theta$. $\Gamma \succeq_\theta \Gamma'$ means that $\Gamma(\mathrm{sp}) \succeq_\theta \Gamma'(\mathrm{sp})$ and $\Gamma(r) = \Gamma'(r)$ for every r such that $\Gamma'(r) = \tau_{\theta'}$ and $\theta' \subseteq \theta$. The symbol Q is used in addition to P to denote programs when comparing two executions.

Lemma 3 (High Step): If $P = (H, R, I)_{\kappa}$, $SL(\kappa) \underline{\not\subseteq} \theta, \Psi; \Gamma \vdash P$, then either:

1. there exists Γ_1 and $P_1 = (H_1, R_1, I_1)_{\kappa}$, such that $P \mapsto P_1$, $\Psi; \Gamma_1 \vdash P_1$, $\Gamma \succeq_{\theta} \Gamma_1$, and $\Psi; \Gamma_1 \vdash P \approx_{\theta} P_1$, or

2. I is of the form (raise κ'; I') or (lower w).

Proof sketch: By case analysis on the first instruction of I. I cannot be `halt`, because the typing rule for `halt` requires the context to be empty. If I is not `halt`, `raise` or `lower`, by the operational semantics and inversion on the typing rules, one can find Γ_1 and P_1 for the next step. The typing rules prohibits writing into a low heap cell, hence low heap cells remain the same after the step. When a register is updated, Γ_1 gives it an updated type whose security label takes $SL(\kappa)$ into account, hence that register or stack slot has a high type in Γ_1. As a result, $\Gamma \succeq_{\theta} \Gamma_1$, *and* $\Psi; \Gamma_1 \vdash P \approx_{\theta} P_1$.

Lemma 4 (Context Discharge): If $P = (H, R, I)_{\theta \triangleright w}$, $\theta \underline{\not\subseteq} \theta'$, $\Psi; \Gamma \vdash P$, $P \mapsto^* (H_0, R_0, \text{halt}[\sigma])$., then there exists Γ' and $P' = (H', R', \text{lower } w)_{\theta \triangleright w}$ such that $\Psi; \Gamma' \vdash P'$, $P \mapsto^* P'$, $\Gamma \succeq_{\theta'} \Gamma'$, *and* $\Psi; \Gamma' \vdash P \approx_{\theta} P'$.

Proof sketch: By generalized induction on the number of steps of the derivation $P \mapsto^* (H_0, R_0, \text{halt}[\sigma])$.

[00123] The base step of zero step is not possible, because the security contexts do not match. In the inductive case, suppose the execution consists of n steps, and the proposition holds for any step number less than n. There are two cases to consider, following Lemma 3.

[00124] In the case where the first instruction of I is not `raise` or `lower`, by Lemma 3, there exists Γ_1 and P_1 such that $P \mapsto P_1$, $\Psi; \Gamma_1 \vdash P_1$, $\Gamma \succeq_{\theta'} \Gamma_1$, $\Psi; \Gamma_1 \vdash P \approx_{\theta'} P_1$, and the security context of P_1 is the same as that of P. Note that P_1 is a step in between P and the final program $(H_0, R_0, \text{halt}[\sigma])$. because the operational semantics is deterministic. Hence by induction hypothesis on P_1, there exists Γ' and P' such that $\Psi; \Gamma' \vdash P'$, $P_1 \mapsto^* P'$, $\Gamma_1 \succeq_{\theta'} \Gamma'$, and $\Psi; \Gamma' \vdash P_1 \approx_{\theta'} P'$. Putting the

above together, $P \mapsto {}^*P', \Gamma \succeq_{\theta'} \Gamma'$ because $\succeq_{\theta'}$ is transitive by definition, and $\Psi;\Gamma' \vdash P \approx_{\theta'} P'$ by definition and the fact that $\Gamma_1 \succeq_{\theta'} \Gamma'$.

[00125] Case $I = \texttt{raise}\,\theta_1 \triangleright w_1; I_1$. By definition of the operational semantics, $P \mapsto P_1$ where $P_1 = (H, R, I_1)_{\theta_1 \triangleright w_1}$. By inversion on $\Psi;\ \Gamma \vdash P$ and the typing rule of `raise`, $\theta \subseteq \theta_1$ and $\Psi;\Gamma;\theta_1 \triangleright w_1 \vdash I_1$. By definition of well-typed programs, $\Psi;\Gamma \vdash P_1$. By induction hypothesis on P_1, there exists Γ_2 and $P_2 = (H_2, R_2, \text{lower } W_1)_{\theta_1 \triangleright w_1}$ such that $\Psi;\Gamma_2 \vdash P_2$, $P_1 \mapsto {}^*P_2, \Gamma \succeq_{\theta'} \Gamma_2$ and $\Psi;\Gamma_2 \vdash P_1 \approx_{\theta'} P_2$. $\Psi;\Gamma_2 \vdash P \approx_{\theta'} P_2$ then follows because the heap and register file remains the same in P and P_1.

[00126] Furthermore, by the operational semantics, $P_2 \mapsto P_3$ where $P_3 = (H_2, R_2, I_3)_\kappa$ and I_3 is the instantiated code of w_1 whose security context is κ. By inversion on the well-typedness of I (*i.e.*, `raise` $\theta_1 \triangleright w_1; I_1$), $\kappa = \theta \triangleright w$. By induction hypothesis on P_3, there exists Γ' and $P' = (H', R', \text{lower } w)_{\theta \triangleright w}$ such that $\Psi;\Gamma' \vdash P', P_3 \mapsto {}^*P', \Gamma_2 \succeq_{\theta'} \Gamma'$, and $\Psi;\Gamma' \vdash P_3 \approx_{\theta'} P'$. Putting the above together, the original proposition holds for case $I = \text{raise}\,\theta_1 \triangleright w_1; I_1$.

[00127] Case $S = \texttt{lower}\ w_1$. By inversion on the typing rule of `lower`, $w = w_1$. Let $P' = P$, the proposition holds.

Lemma 5 (Low Step): If $P = (H, R, I)_\kappa, \text{SL}(\kappa) \subseteq \theta, \Psi;\Gamma \vdash P$, $\Psi;\Gamma \vdash Q$, $\Psi;\Gamma \vdash P \approx_\theta Q$, $P \mapsto P_1$, $Q \mapsto Q_1$, then exists Γ_1 such that $\Psi;\Gamma \vdash P_1, \Psi;\Gamma \vdash Q_1$, and $\Psi;\Gamma_1 \vdash P_1 \approx_\theta Q_1$.

Proof sketch: By case analysis on the first instruction of I. Since $SL(\kappa) \subseteq \theta, P$ and Q contains the same instruction sequence by definition of $\approx_\theta$. The case of raising to a higher context does not change the state, thereby trivially maintaining the equivalence. All other cases maintain that the security context is lower than θ. Inspection on the typing derivation shows that low locations in the heap can only be assigned low values. Once a register is given a high value, its type in Γ_1 will change to high. In the case of branching, the guard must be low, so both P and Q

branch to the same code. Hence the two programs remain equivalent after one step.

Theorem 1 (Noninterference): If $P = (H, R, I)_{\kappa}, SL(\kappa) \subseteq \theta', \Psi;\Gamma \vdash P, \Psi;\Gamma \vdash Q,$ $\Psi;\Gamma \vdash P \approx_{\theta} Q, P \mapsto^{*} (H_p, R_p, \text{halt}[\sigma_p])., \text{and } Q \mapsto^{*} (H_q, R_q, \text{halt}[\sigma_q])., \text{then}$ exists Γ' such that $\Psi;\Gamma' \vdash (H_p, R_p, \text{halt}[\sigma_p])._{\cdot} \approx_{\theta} (H_q, R_q, \text{halt}[\sigma_q])._{\cdot}$

Proof sketch: By generalized induction on the number of steps of the derivation $P \mapsto^{*} (H_p, R_p, \text{halt}[\sigma_p])..$ The base case of zero step is trivial. The inductive case is done by case analysis on the first instruction of I.

Consider the case where I is of the form `raise` $\theta_1 \triangleright w_1; I_1$ where $\theta_1 \not\subseteq \theta$. By definition of the operational semantics and the typing rules, $P \mapsto P_1$ where $= P_1 = (H, R, I_1)_{\theta_1 \triangleright w_1}$ and $\Psi;\Gamma \vdash P_1$. By Lemma 4, there exists Γ_2 and $P_2 = (H_2, R_2,$ lower $w_1)_{\theta_1 \triangleright w_1}$ such that $\Psi;\Gamma_2 \vdash P_2, P_1 \mapsto^{*} P_2, \Gamma \succeq_{\theta} \Gamma_2, \text{and} \Psi;\Gamma_2 \vdash P_1 \approx \theta P_2.$ Hence $\Psi; \vdash H \approx_{\theta} H_2$ and $\Psi;\Gamma_2 \vdash R \approx_{\theta} R_2$.

[00128] By the operational semantics, $P_2 \mapsto P_3$ where $w_1 = l_1[\vec{\psi}]$ $P_3 = (H_2, R_2, I_3[\vec{\psi}/\Delta])_{\kappa 3}$ and $H(l_1) = \text{code}[\Delta]\langle\kappa_3\rangle\Gamma_3.I_3$. By inversion on the typing derivation of $\Psi;\Gamma_2 \vdash P_2$, $\Gamma_3 \subseteq \Gamma_2$ and $\Psi;\Gamma_3 \vdash P_3$. it follows that $\Psi;\Gamma_3 \vdash R \approx_{\theta} R_2$. By inversion on the typing derivation of $\Psi;\Gamma \vdash P$ where the first instruction of P is `raise` $\theta_1 \triangleright w_1; I_1, \kappa_3 = \kappa$.

[00129] By similarly reasoning, $Q \mapsto^{*} Q_3$ where $Q_3 = (H'_2, R'_2, I_3)_{\kappa 3}, \Psi \vdash$ $H \approx_{\theta} H'_2,$

$\Psi;\Gamma_3 \vdash R \approx_{\theta} R'_2$ and $\Psi;\Gamma_3 \vdash Q_3$. By transitivity of the equivalence relations, $\Psi \vdash$ $H_2 \approx_{\theta} H'_2$

and $\Psi;\Gamma_3 \vdash R_2 \approx_{\theta} R'_2$. Hence $\Psi;\Gamma \vdash P_3 \approx_{\theta} Q_3$. The case then follows by induction hypothesis.

[00130] All other cases remain low after a step. By Lemma 5, the two executions in the next step are equivalent and well typed. The proof of these cases then follows by induction

hypothesis.

Certifying Compilation For Confidentiality

[00131] The noninterference theorem described above guarantees that well-typed TAL_C programs satisfy the information flow policy, even in the presence of memory aliasing and first-class code pointers. The following describes how TAL_C may serve as the target of certifying compilation (Figures 17A, 17B and 17C).

[00132] Certifying compilation for a realistic language typically involves a complex sequence of transformations, including CPS and closure conversion, heap allocation, and code generation. A simple security-type system of Figure 2 is chosen as a source language. This allows a concise presentation, yet suffices in demonstrating the separation of security-context operations `raise` and `lower` from conventional instructions and mechanisms (e.g., stack convention for procedure calls).

[00133] The low-high security hierarchy of Figure 2 defines a simple lattice consisting of two elements: $\bot$ and $\top$. In the following, $|t|$ is used to denote the translation of source type t in TAL_C: $|low| \equiv \text{int}_\bot$ and $|high| \equiv \text{int}_\top$. The procedure types are also translated from the source language into TAL_C as follows:

$$|\langle \mathtt{pc} \rangle (\mathtt{t}_1, \ldots, \mathtt{t}_n) \rightarrow \mathtt{void}| = (\forall[\Delta].\langle \kappa \rangle \{\mathtt{sp} : \Sigma\})_\bot$$
$$\text{where } (\Delta, \kappa) = \begin{cases} (\rho_\mathtt{o}, \bullet) & \text{if } \mathtt{pc} = \mathtt{low} \\ (\alpha\rho_\mathtt{o}, \top \rhd \alpha) & \text{if } \mathtt{pc} = \mathtt{high} \end{cases}$$
$$\text{and } \Sigma = (\forall[\Delta].\langle \kappa \rangle \{\mathtt{sp} : \rho\})_\bot :: \langle |\mathtt{t}_1| \rangle_\bot :: \ldots :: \langle |\mathtt{t}_n| \rangle_\bot :: \rho$$

[00134] This procedure type translation assumes a calling convention where the caller pushes a return pointer and the location of the arguments (implementing the call-by-reference semantics of the source language) onto the stack, and the callee deallocates the current stack frame upon return. The stack type Σ refers to a variable ρ because the procedure may be called under different stacks, as long as the current stack frame is as expected. The security context κ is empty if pc is low, or $\top \rhd \alpha$ if pc is high. Postdominator variable α is used because the procedure

may be called in security contexts with different postdominators. The type environment Δ simply collects all the needed type variables.

[00135] The program translation starts in a heap H_0 and a heap type Ψ_0 which satisfy $\vdash H_0 : \Psi_0$ and contain entries for all the variables and procedures of the source program. For any source variable v that $\Phi(v)=t$, there exists a location l_v in the heap such that $\Psi_0(l_v) = \langle |t| \rangle_\perp$. For any source procedure f that, $\Phi(f)=\langle pc \rangle(t_1, ..., t_n) \rightarrow void$, there exists a location l_f in the heap such that $\Psi_0(l_f)=|\langle pc \rangle(t_1, ..., t_n) \rightarrow void\ |$. $\Phi \sim \Psi_0$ is used to refer to this correspondence.

[00136] In one embodiment, the above heap Ψ_0 can be constructed with dummy slots for the procedures---the code in there simply jumps to itself. This suffices for typing the initial heap, thus facilitating the type-preservation proof. It creates locations for all source procedures and allows the translation of the actual code to refer to them.

[00137] The translation details are given in Figures 17A, 17B and 17C, based on the structure of the typing derivation of the source program. Which translation rule to apply is determined by the last typing rule used to check the source construct (program, procedure, or command). We use TD to denote (possibly multiple) typing derivations.

[00138] An expression translation of the form $|E| = \vec{\iota} \parallel v$ is defined in Figure 17A. The instruction vector $\vec{\iota}$ computes the value of E and the result is put in the register r. For a global variable, the value is loaded from the heap using its corresponding heap label. For a procedure argument, the location of the actual entity is loaded from the stack, and the value is then loaded from the heap.

[00139] In Figure 17B, when translating a program (Rule [TRP1]), all the procedure declarations are translated, add halting code as the ending point of the program, and proceed to translate the main command. The result triple contains the updated heap type and heap, and a starting label l which leads to the starting point of the program. Procedure translation (Rule [TRF1]) takes care of part of the calling convention. It adds epilogue code that loads the return pointer, deallocates the current stack frame and transfers the control to the return pointer. It then

resorts to command translation to translate the procedure body, providing the label to the epilogue code as the ending point of the procedure body.

[00140] Figure 17C define command translation of the form

$$\left\lfloor \frac{TD}{[pc] \vdash C} \right\rfloor \left[\begin{array}{l} \Psi \\ H \\ l_{start}; l_{end}; \Delta; \kappa; \Sigma \end{array} \right] = \left[\begin{array}{l} \Psi' \\ H' \end{array} \right].$$

[00141] This command translation takes 7 arguments: a code heap type (Ψ), a code heap (H), starting and ending labels (l_{start} and l_{end}) for the computation of C, a type environment (Δ), a security context (κ), and a stack type (Σ). It generates the extended code heap type (Ψ') and code heap (H'). Unsurprisingly, this translation appears complex, because it provides a formal model of a certifying compiler. Nonetheless, it is easy to follow if some invariants maintained by the translation are remembered:

- H is well-typed under Ψ and contains entries for all source variables and procedures;
- Ψ and H already contain the continuation code labeled l_{end};
- The new code labeled l_{start} will be put in Ψ' and H';
- The security context κ must match pc;
- The stack type Σ contains entries for all procedure arguments, if the command being compiled is in the body of a procedure;
- The environment Δ contains all free type variables in κ and Σ.

[00142] Most of the command translation rules simply put Δ, κ and Σ in place for the generated code types, and further propagate them to the translation of sub-components. The only rule that non-trivially manipulates the security context is Rule [TRC4]--when a subsumption rule is used for typing a source command, the translation generates code that is enclosed in a `raise-lower` pair. The translation of the sub-component is carried out in an updated heap with a new ending label l_1. The code at l_1 restores the security context and transfers the control to the given ending label l'. After the translation of the sub-component,

code is added at the starting label l to raise the security context to the expected level.

[00143] Procedure call translation is given as Rule [TRC7]. It creates "prologue" code that allocates a stack frame, pushes the return pointer and the arguments onto the stack, and jumps to the procedure label. Note that the corresponding epilogue code is generated by the procedure declaration translation in Rule [TRF1].

[00144] The translation of while-loops is also interesting (Rule [TRC6]). When translating the loop body, the continuation block needs to be prepared, which happens to be the code for the loop test. A dummy block labeled l is used to serve as the continuation block when translating the body C. This block is introduced for maintaining the above invariants. It facilitates the type-preservation proof of the translation. After the translation of the loop body, this dummy block is replaced with the actual code that implements the loop test, as shown on the bottom right side of Rule [TRC6].

Lemma 6 (Expression Translation) If $\Phi \sim \Psi$, $\Phi \vdash \mathtt{E} : \mathtt{t}$, $|\mathtt{E}| = \vec{\iota} \,||\, r$, and $\Psi; \Delta; \{r : |\mathtt{t}|, \mathtt{sp} : \Sigma\}; \kappa \vdash I$, then $\Psi; \Delta; \{\mathtt{sp} : \Sigma\}; \kappa \vdash \vec{\iota}; I$.

Lemma 7 (Command Translation) If $\Phi \sim \Psi$, $\Phi; [\mathtt{pc}] \vdash \mathtt{C}$,

$$\left|\frac{\mathtt{TD}}{\Phi; [\mathtt{pc}] \vdash \mathtt{C}}\right| \left[\begin{array}{l} \Psi \\ H \\ l_{start}; l_{end}; \Delta; \kappa; \Sigma \end{array}\right] = \left[\begin{array}{l} \Psi' \\ H' \end{array}\right],$$

$\Psi(l_{end}) = (\forall[\Delta].\langle\kappa\rangle\,\{\mathtt{sp} : \Sigma\})_\perp$, $SL(\kappa) = |\mathtt{pc}|$, $\vdash H : \Psi$, then $\Phi \sim \Psi'$, $\vdash H' : \Psi'$ and $\Psi'; \Delta \vdash l_{start} : (\forall[\Delta].\langle\kappa\rangle\,\{\mathtt{sp} : \Sigma\})_\perp$.

[00145] The proofs for the above two lemmas are straightforward by structural induction on the derivation of the translation. Type preservation of procedure translation can be derived from Lemma 7 based on Rule [TRF1]. Type preservation of program translation then follows based on Rule [TRP1].

Lemma 8 (Procedure Translation) If $\Phi \sim \Psi$, $\Phi \vdash \mathtt{F}$, $\vdash H : \Psi$,

$$\left|\frac{\mathtt{TD}}{\Phi \vdash \mathtt{F}}\right| \left[\begin{array}{l} \Psi \\ H \end{array}\right] = \left[\begin{array}{l} \Psi' \\ H' \end{array}\right], \text{ then } \Phi \sim \Psi' \text{ and } \vdash H' : \Psi'.$$

Theorem 2 (Program Translation) If $\Phi \sim \Psi_0$, $\Phi \vdash \mathtt{P}$, $\vdash H_0 : \Psi_0$, $\left\lfloor \frac{\mathtt{TD}}{\Phi \vdash \mathtt{P}} \right\rfloor \left[\begin{array}{c} \Psi_0 \\ H_0 \end{array}\right] = \left[\begin{array}{c} \Psi \\ H; l \end{array}\right]$, then $\Phi \sim \Psi$ and $\Psi; \{\mathtt{sp} : nil\} \vdash (H, \{\mathtt{sp} : nil\}, \mathtt{jmp}\ l)_\bullet$.

Extensions and Alternatives

[00146] Orthogonal Features: In the above discussions, TAL_C focuses on a minimal set of language features. In alternative embodiments, polymorphic and existential types, as seen in TAL, are orthogonal and can be introduced with little difficulty. Furthermore, since TAL_C is compatible with TAL, it is also possible to accommodate other features of the TAL family. For instance, alias types may provide a more accurate alias analysis, improving the current conservative approach that considers every pointer as a potential alias. In the following, we will also discuss the use of singleton types.

[00147] Security Polymorphism: TAL_C relies on a security context $\theta \triangleright w$ to identify the current security level θ and its ending point w. It is monomorphic with respect to security, because the security context of a code block is fixed. In practice, security-polymorphic code can also be useful.

[00148] Figure 18 gives an example. The function `double` can be invoked with either low or high input. It is safe to invoke `double` in a context if only the security level of the input matches that of the context. In a polymorphic TAL_C-like type system, `double` can be given the type $(\forall[\theta,\alpha].\langle\theta \triangleright \alpha\rangle\{r_1 : \mathrm{int}_\theta, r_0 : (\forall[].\langle\theta \triangleright \alpha\rangle\{r_1 : \mathrm{int}_\theta\})_\perp\})_\perp$. In this case, r_1 is the argument register, r_0 stores the return pointer, and meta-variables θ is reused as a variable.

[00149] It is straightforward to support this kind of polymorphism. In fact, most of the required constructs are already present in TAL_C. We omitted such polymorphism simply because it complicates the presentation without providing additional insights. Nonetheless, the expressiveness of such polymorphism is still limited. Since the label α is not known until instantiated, the code of `double` has no knowledge about α. Hence the security context $\theta \triangleright \alpha$ cannot be discharged within the body of `double`.

[00150] It is not obvious why one would wish to discharging the security context within a polymorphic function. Indeed, it is always possible to wrap a function call inside a secured region by symmetric `raise` and `lower` operations from the caller's side. However, the asymmetric discharging of security context may sometimes be desirable for *certifying optimization.* For instance, in Figure 18, `double` is called as the last statement of the body of a high conditional. In this case, directly discharging the security context when `double` returns would remove a superfluous lower operation from the caller's side. Such a discharging requires `lower` to operate on small values (in particular, registers)—since the return label is not static, it is passed in through a register.

[00151] It may require singleton types and intersection types to support such a powerful `lower` operation. For example, a double function that automatically discharges its
security context can have the type

$$\left(\forall[\theta,\alpha].\langle\theta \triangleright \alpha\rangle \left\{\begin{array}{l} r_1 : \mathtt{int}_\theta, \\ r_0 : \mathtt{sint}(\alpha)_\perp \wedge (\forall[].\langle\bullet\rangle\{r_1 : \mathtt{int}_\theta\})_\perp \end{array}\right\}\right)_\perp .$$

[00152] At the end of the function, an instruction `lower` r_0 discharges the security context and transfers the control to the return code. For type checking, the singleton integer type **sint**(α) matches the register r_0 with the label in the security context, and the code type ensures that the control flow to the return code is safe.

[00153] Full erasure: With the powerful type constructs discussed above, one can achieve a full erasure for the `lower` operation. Instead of treating `lower` as an instruction, one can treat it s a transformation on small values. This is in spirit similar to the *pack* operation of existential types in TAL. Such a lower transformation bridges the gap between the current security context and the security level of the target label. The actual control flow transfer is then completed with a conventional jump instruction (*e.g.*, `jmp (lower` r_0`)`). One can also achieve a full erasure for `lower` even without dependent types. The idea is to separate the jump instruction into direct jump and indirect jump. This is also consistent with real machine architectures. The `lower` operation, similar to *pack*,

transforms word values (eventually, direct labels). Lowered labels, similar to packed values, may serve as the operand of direct jump. Indirect jump, on the other hand, takes normal small values. This is expressive enough for certifying compilation, yet may not be sufficient for certifying optimization as discussed above.

An Exemplary Mobile Phone

[00154] Figure 19 is a block diagram of one embodiment of a cellular phone. Referring to Figure 19, the cellular phone 1910 includes an antenna 1911, a radio-frequency transceiver (an RF unit) 1912, a modem 1913, a signal processing unit 1914, a control unit 1915, an external interface unit (external I/F) 1916, a speaker (SP) 1917, a microphone (MIC) 1918, a display unit 1919, an operation unit 1920 and a memory 1921. These components and their operation are well-known in the art.

[00155] In one embodiment, control unit 1915 includes a CPU (Central Processing Unit), which cooperates with memory 1921 to perform the operations described above.

An Exemplary Computer System

[00156] Figure 20 is a block diagram of an exemplary computer system that may perform one or more of the operations described herein. Referring to Figure 20, computer system 2000 may comprise an exemplary client or server computer system. Such a client may be part of another device, such as a mobile device.

[00157] Computer system 2000 comprises a communication mechanism or bus 2011 for communicating information, and a processor 2012 coupled with bus 2011 for processing information. Processor 2012 includes a microprocessor, but is not limited to a microprocessor, such as, for example, Pentium™, PowerPC™, etc.

[00158] System 2000 further comprises a random access memory (RAM), or other dynamic storage device 2004 (referred to as main memory) coupled to bus 2011 for storing information and instructions to be executed by processor 2012.

Main memory 2004 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 2012.

[00159] Computer system 2000 also comprises a read only memory (ROM) and/or other static storage device 2006 coupled to bus 2011 for storing static information and instructions for processor 2012, and a data storage device 2007, such as a magnetic disk or optical disk and its corresponding disk drive. Data storage device 2007 is coupled to bus 2011 for storing information and instructions.

[00160] Computer system 2000 may further be coupled to a display device 2021, such as a cathode ray tube (CRT) or liquid crystal display (LCD), coupled to bus 2011 for displaying information to a computer user. An alphanumeric input device 2022, including alphanumeric and other keys, may also be coupled to bus 2011 for communicating information and command selections to processor 2012. An additional user input device is cursor control 2023, such as a mouse, trackball, trackpad, stylus, or cursor direction keys, coupled to bus 2011 for communicating direction information and command selections to processor 2012, and for controlling cursor movement on display 2021.

[00161] Another device that may be coupled to bus 2011 is hard copy device 2024, which may be used for marking information on a medium such as paper, film, or similar types of media. Another device that may be coupled to bus 2011 is a wired/wireless communication capability 2025 to communication to a phone or handheld palm device.

Note that any or all of the components of system 2000 and associated hardware may be used in the present invention. However, it can be appreciated that other configurations of the computer system may include some or all of the devices.

[00162] Whereas many alterations and modifications of the present invention will no doubt become apparent to a person of ordinary skill in the art after having read the foregoing description, it is to be understood that any particular embodiment shown and described by way of illustration is in no way intended to be considered limiting. Therefore, references to details of various

embodiments are not intended to limit the scope of the claims which in themselves recite only those features regarded as essential to the invention.

CLAIMS

We claim:

1. A method comprising:

receiving securely typed native code;

performing verification with respect to information flow for the securely typed native code based on a security policy.

2. An article of manufacture having one or more recordable media storing instructions thereon which, when executed by a system, causes the system to perform a method comprising:

receiving securely typed native code;

performing verification with respect to information flow for the securely typed native code based on a security policy.

3. An apparatus comprising:

a memory to store annotated assembly code, a verification module, an code modification module; and

a processor to execute the verification module to perform verification with respect to information flow for the annotated code based on a security policy.

4. A method comprising:

performing a security-type preserving translation on assembly code, including

annotating memory, stack and register contents with security levels, and

rebuilding source-level structure of the assembly code with annotations by adding operations to the assembly code to mark the

beginning and ending of a security region of the assembly code in which two execution paths based on confidential information are encountered;

certifying compilation for information flow resulting from the assembly code when executed; and

sending the securely typed assembly code onto a network.

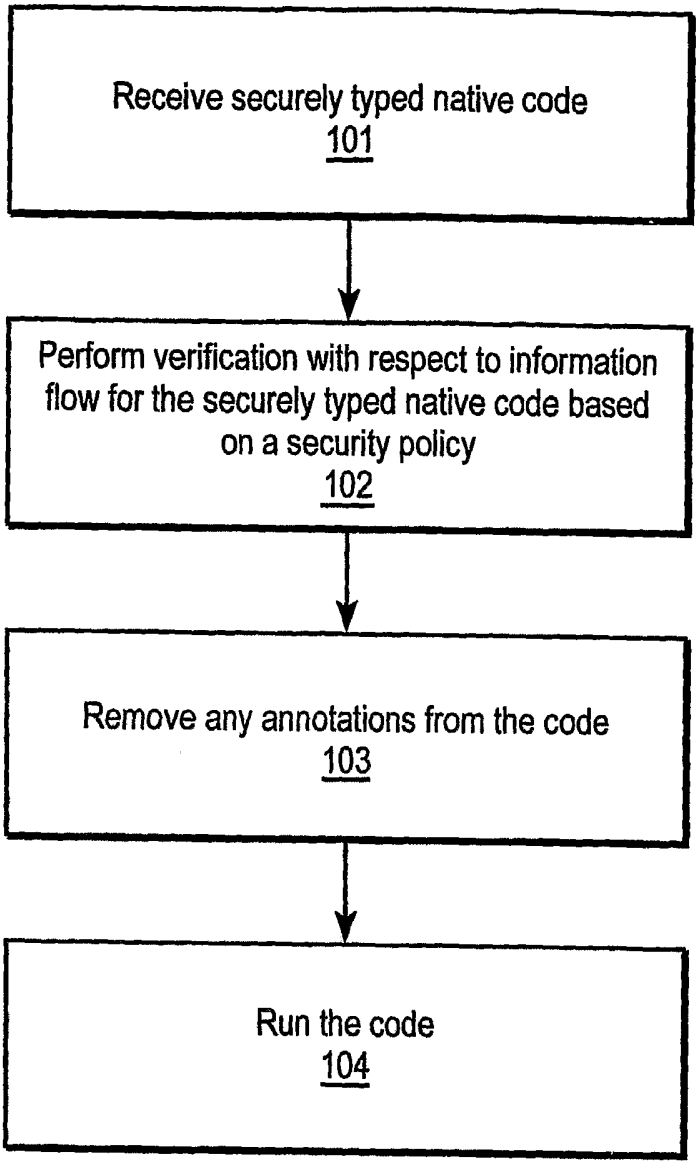
Receive securely typed native code
101
Perform verification with respect to information flow for the securely typed native code based on a security policy
102
Remove any annotations from the code
103
Run the code
104

FIG. 1A

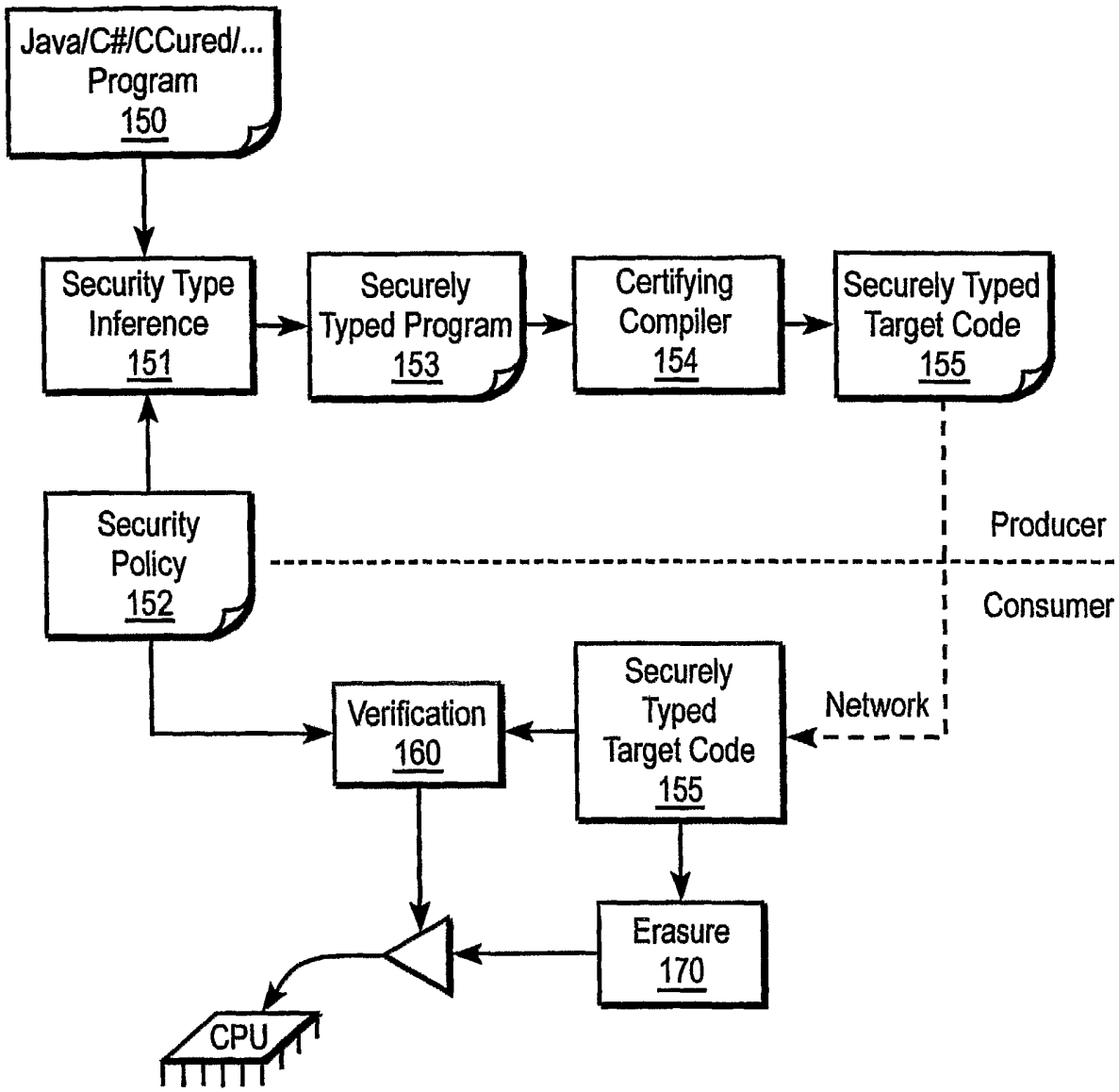
Java/C#/CCured/...
Program
150
Security Type
Inference
151
Securely
Typed Program
153
Certifying
Compiler
154
Securely Typed
Target Code
155
Security
Policy
152
Producer
Consumer
Verification
160
Securely
Typed
Target Code
155
Network
Erasure
170
CPU

FIG. 1B

(*Typ*) $t, pc ::= low \mid high$

(*Var*) $V ::= v \mid x$

(*Env*) $\Phi ::= o \mid V: t, \Phi \mid f: \langle pc \rangle (t_1,...,t_n) \rightarrow void, \Phi$

(*Exp*) $E ::= i \mid V \mid E_1 + E_2$

(*Comm*) $C ::= V := E \mid C_1; C_2 \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C \mid f(V_1,..., V_n)$

(*Fun*) $F ::= f\langle pc \rangle (x_1:t_1, ..., x_n:t_n) \{C\}$

(*Prog*) $P ::= \{F_1;...; F_n;C\}$

[E1–2] $\Phi \vdash E : high \qquad \Phi \vdash i : t$

[E3–4] $\dfrac{\Phi(V) = low}{\Phi \vdash V : low} \qquad \dfrac{\vdash E_1 : low \ \vdash E_2 : low}{\vdash E_1 + E_2 : low}$

[C1–2] $\dfrac{\Phi(V) = high}{\Phi; [pc] \vdash V := E} \qquad \dfrac{\Phi(V) = low \ \ \Phi \vdash E : low}{\Phi; [low] \vdash V := E}$

[C3] $\dfrac{\Phi \vdash E : pc \ \ \Phi; [pc] \vdash C_1 \ \ \Phi; [pc] \vdash C_2}{\Phi; [pc] \vdash \text{if } E \text{ then } C_1 \text{ else } C_2}$

[C4–5] $\dfrac{\Phi; [high] \vdash C}{\Phi; [low] \vdash C} \qquad \dfrac{\Phi; [pc] \vdash C_1 \ \ \Phi; [pc] \vdash C_2}{\Phi; [pc] \vdash C_1; C_2}$

[C6] $\dfrac{\Phi \vdash E : pc \ \ \Phi; [pc] \vdash C}{\Phi; [pc] \vdash \text{while } E \text{ do } C}$

[C7] $\dfrac{\Phi(f) = \langle pc \rangle (t_1,..., t_n) \rightarrow void \qquad \underset{i \in \{1...n\}}{A} \ \Phi \vdash V_i : t_i}{\Phi; [pc] \vdash f(V_1,...,V_n)}$

[F1] $\dfrac{x_1 : t_1,...,x_n : t_n, \Phi; [pc] \vdash C}{\Phi \vdash f\langle pc \rangle (x_1 : t_1,...,x_n : t_n)\{C\}}$

[P1] $\dfrac{\begin{array}{l} F_i = f_i\langle pc_i \rangle (x_{1i} : t_{1i},..., x_{n_i i} : t_{n_i i})\{C_i\} \\ \Phi(F_i) = \langle pc_i \rangle (t_{1i},..., t_{n_i i}) \rightarrow void \end{array} \qquad \Phi \vdash F_i \quad \Phi;[low] \vdash C \quad \underset{i \in \{1...n\}}{A}}{\Phi \vdash \{F_i;...; F_n; C\}}$

FIG. 2

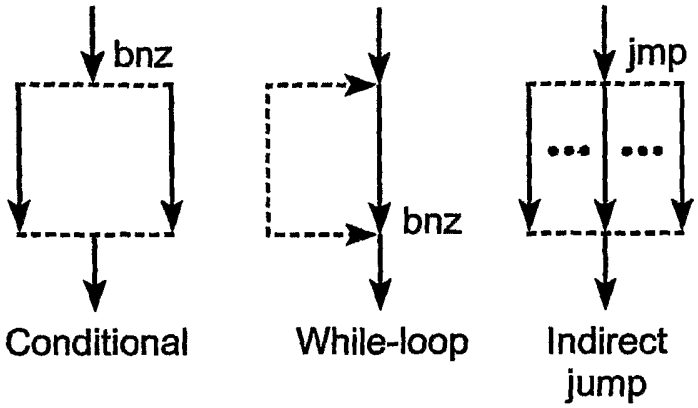

FIG. 3

```
(* suppose p_h alias p_l *)
if (h = 0) then p_h=new cell;
*p_h=1;
(* now *p_l reveals h *)
```

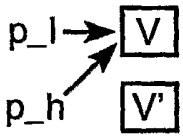

FIG. 4

```
fun f0 ( ) = (1 : =0;  ( ) )
fun f1 ( ) = (1 : =1;  ( ) )
let f = (if h then f1 else f0)
  in f ( )
```

FIG. 5

```
fun f0 ( ) = (h' : =0;  ( ) )
fun f1 ( ) = (h' : =1;  ( ) )

if h then f  :=  f1 else f  :=  f0;
1 : = 1;  !f( );  1 : = 1*2; ...
```

FIG. 6

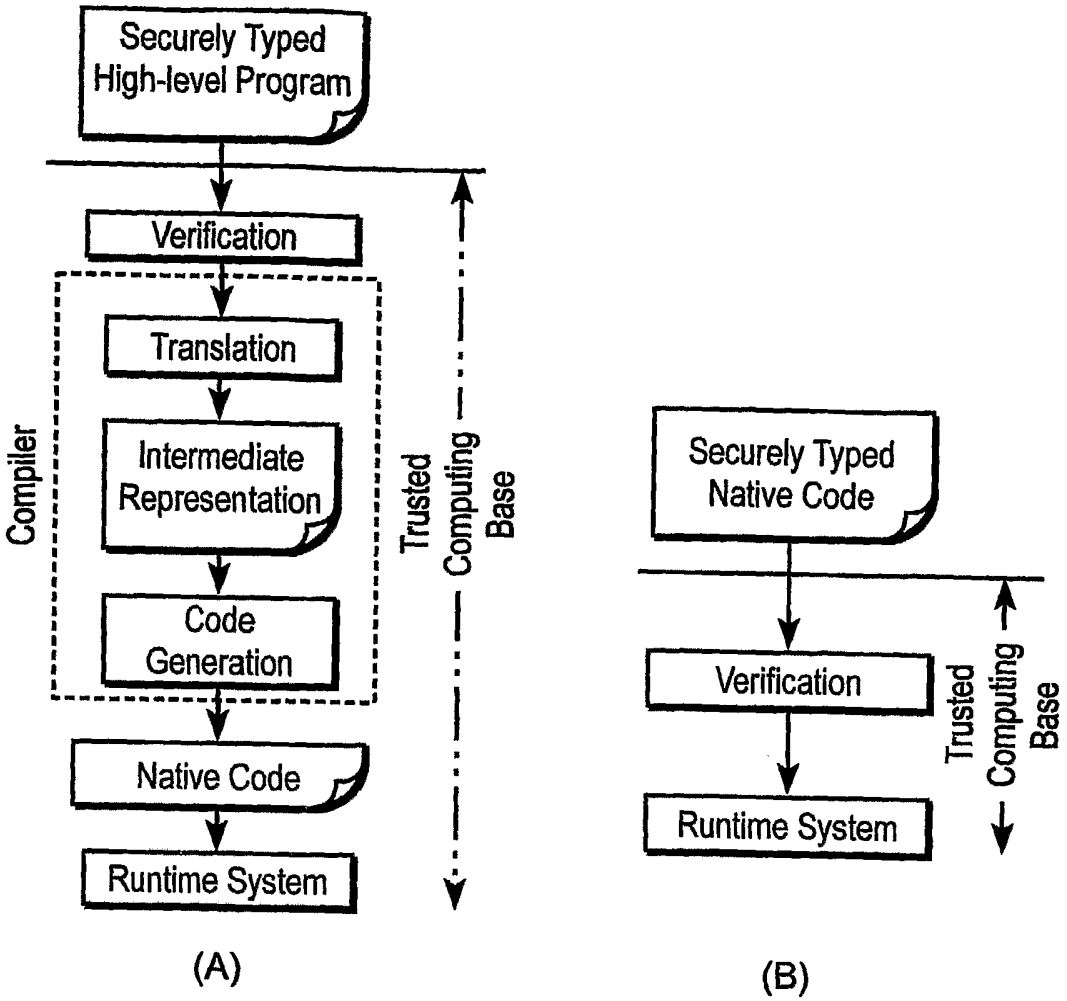
Securely Typed High-level Program
Verification
Translation
Intermediate Representation
Code Generation
Compiler
Native Code
Runtime System
Trusted Computing Base
(A)
Securely Typed Native Code
Verification
Runtime System
Trusted Computing Base
(B)

FIG. 7

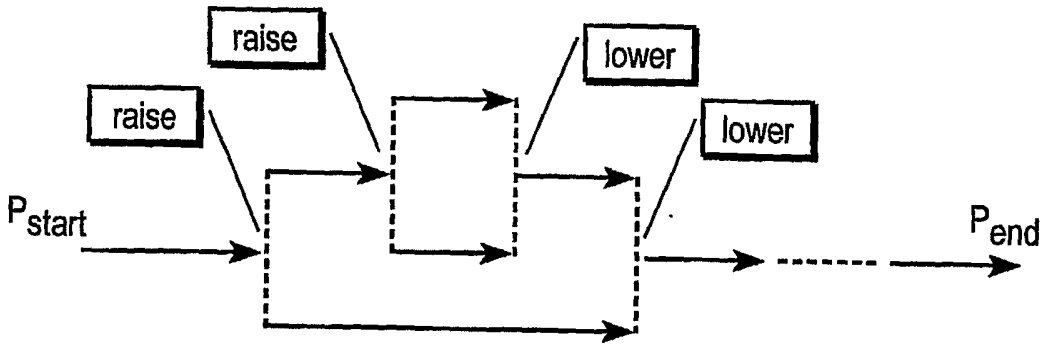
raise
raise
lower
lower
Pstart
Pend

FIG. 8

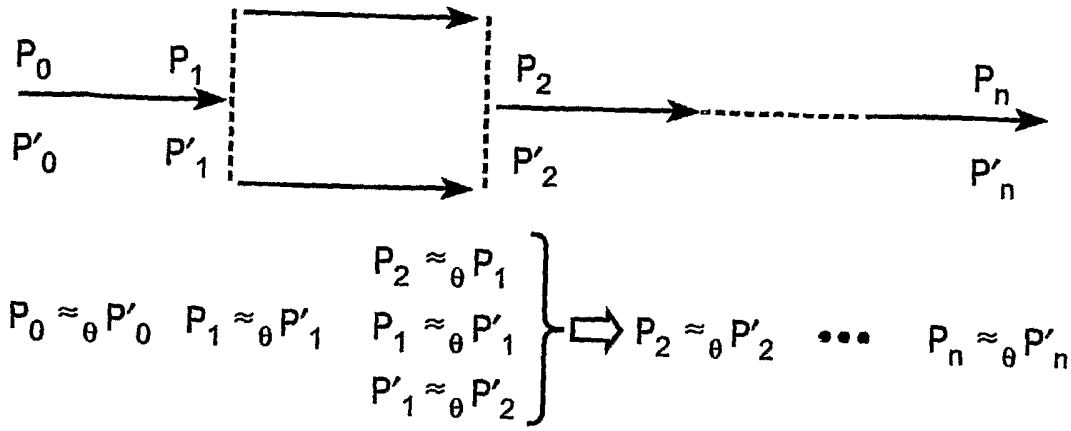
P0
P1
P2
Pn
P'0
P'1
P'2
P'n
P0 ≈θ P'0
P1 ≈θ P'1
P2 ≈θ P1
P1 ≈θ P'1
P'1 ≈θ P'2
P2 ≈θ P'2
Pn ≈θ P'n

FIG. 9

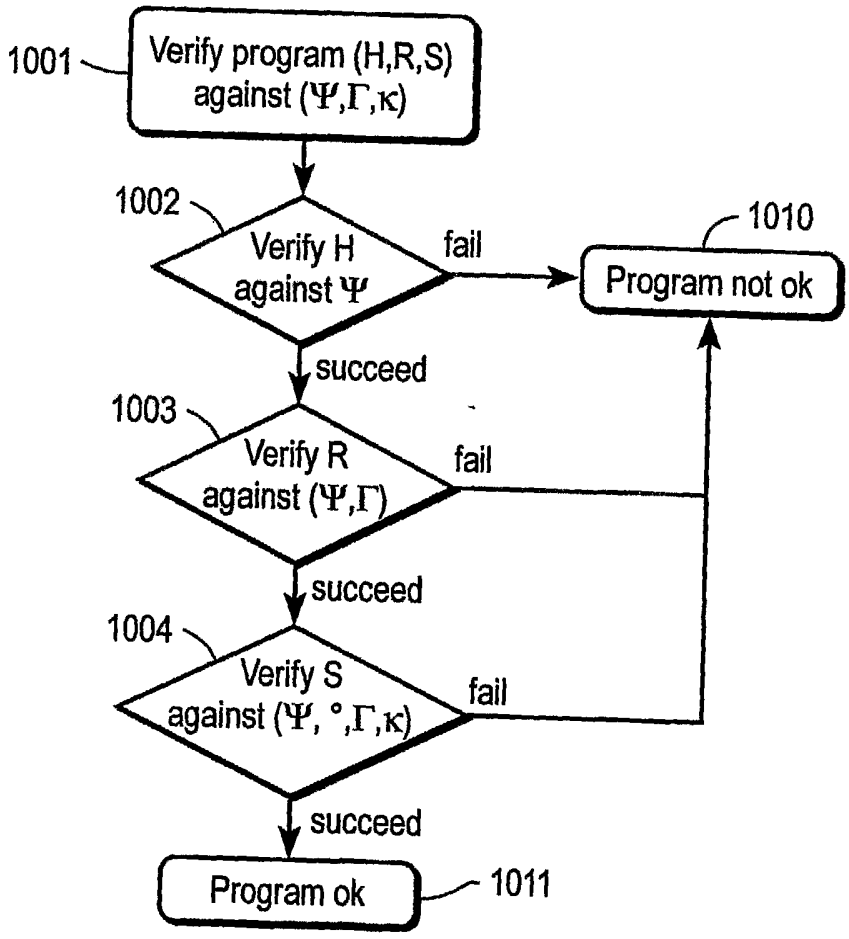
1001
Verify program (H,R,S) against (Ψ,Γ,κ)
1002
Verify H against Ψ
fail
1010
Program not ok
succeed
1003
Verify R against (Ψ,Γ)
fail
succeed
1004
Verify S against (Ψ, °,Γ,κ)
fail
succeed
Program ok
1011

FIG. 10

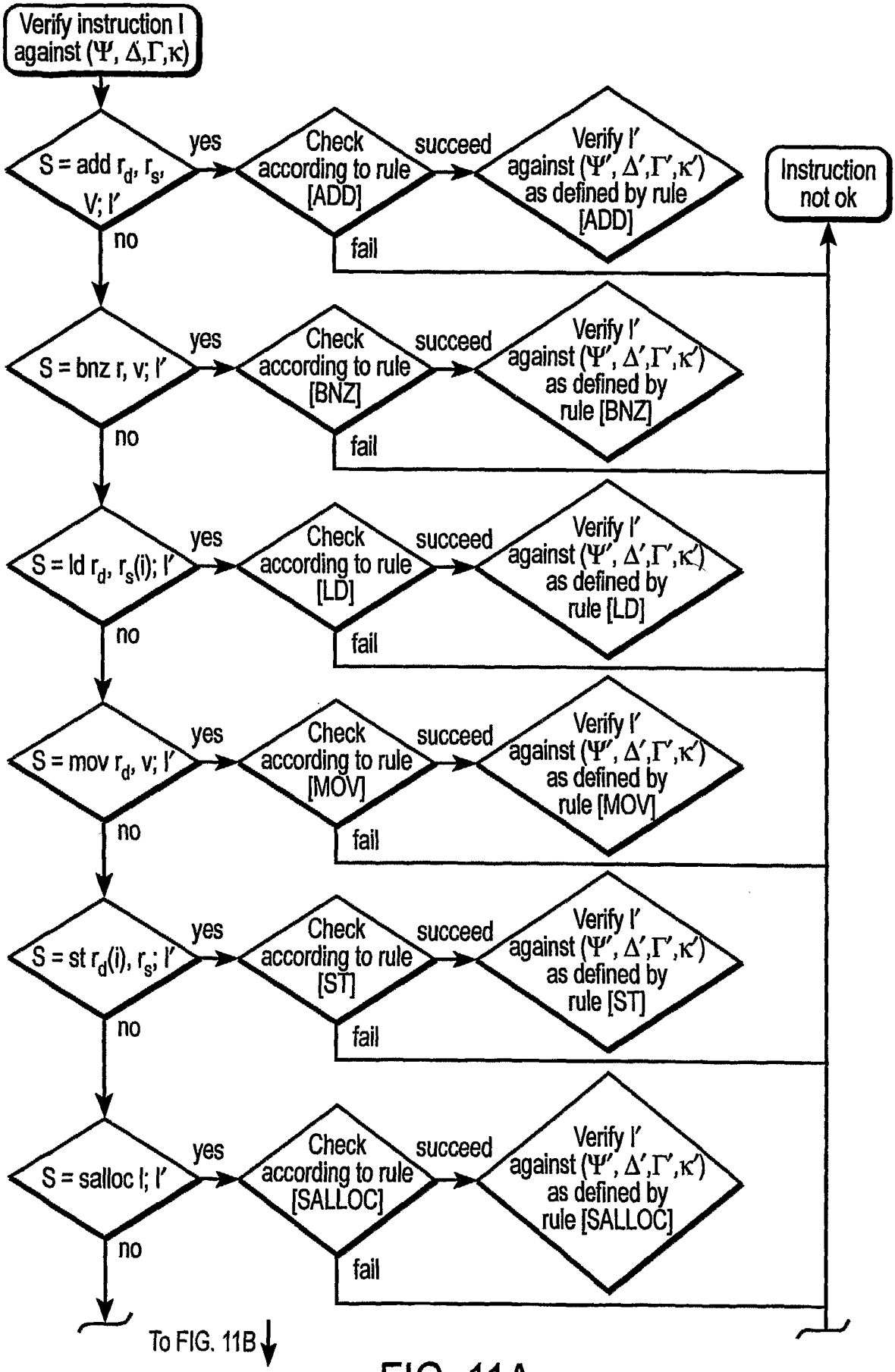
Verify instruction I against (Ψ, Δ,Γ,κ)
S = add r_d, r_s, V; I′
yes
Check according to rule [ADD]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [ADD]
fail
no
S = bnz r, v; I′
yes
Check according to rule [BNZ]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [BNZ]
fail
no
S = ld r_d, r_s(i); I′
yes
Check according to rule [LD]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [LD]
fail
no
S = mov r_d, v; I′
yes
Check according to rule [MOV]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [MOV]
fail
no
S = st r_d(i), r_s; I′
yes
Check according to rule [ST]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [ST]
fail
no
S = salloc I; I′
yes
Check according to rule [SALLOC]
succeed
Verify I′ against (Ψ′, Δ′,Γ′,κ′) as defined by rule [SALLOC]
fail
no
Instruction not ok
To FIG. 11B

FIG. 11A

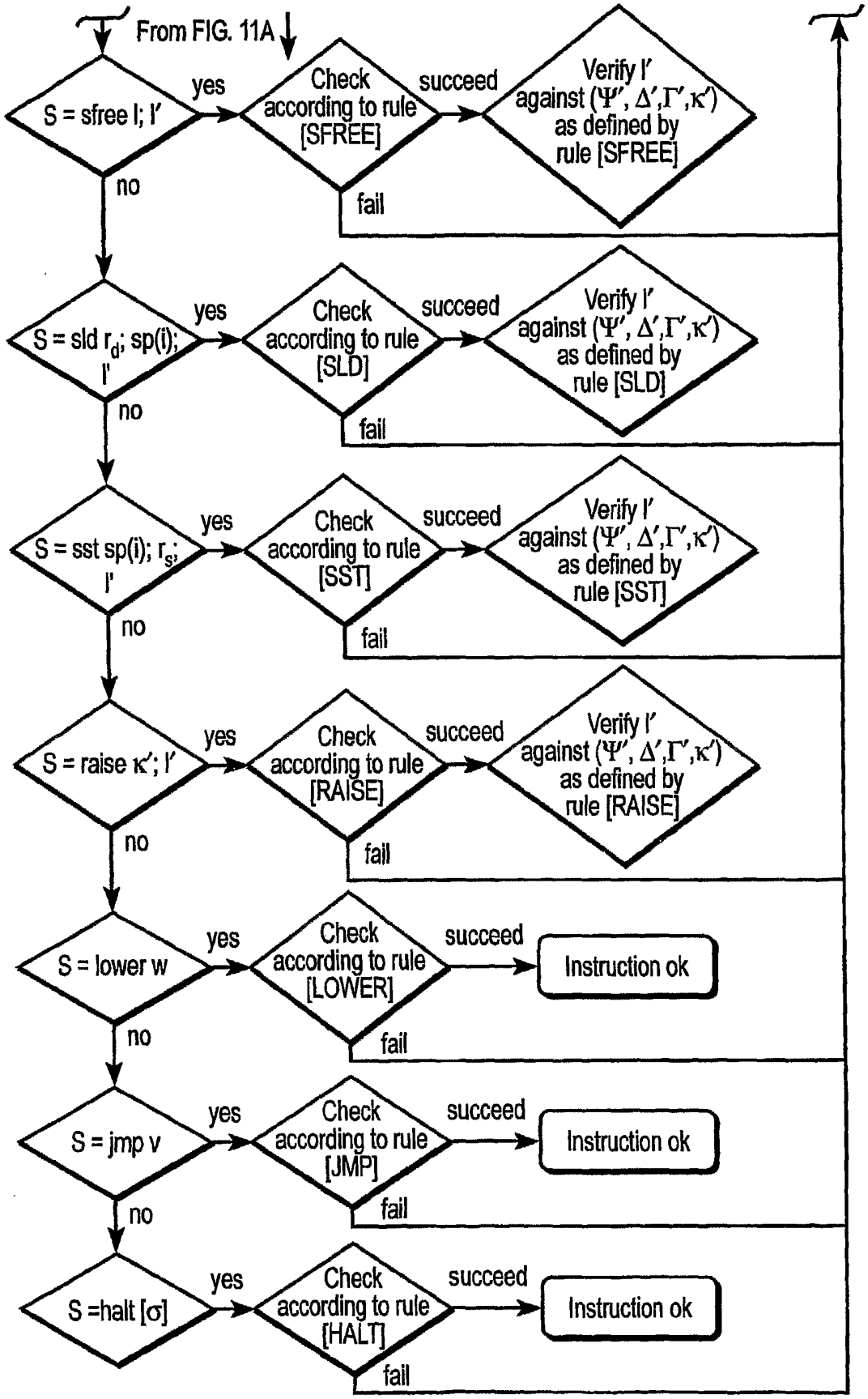
From FIG. 11A
S = sfree l; l'
yes
Check according to rule [SFREE]
succeed
Verify l' against (Ψ', Δ',Γ',κ') as defined by rule [SFREE]
fail
no
S = sld r_d; sp(i); l'
yes
Check according to rule [SLD]
succeed
Verify l' against (Ψ', Δ',Γ',κ') as defined by rule [SLD]
fail
no
S = sst sp(i); r_s; l'
yes
Check according to rule [SST]
succeed
Verify l' against (Ψ', Δ',Γ',κ') as defined by rule [SST]
fail
no
S = raise κ'; l'
yes
Check according to rule [RAISE]
succeed
Verify l' against (Ψ', Δ',Γ',κ') as defined by rule [RAISE]
fail
no
S = lower w
yes
Check according to rule [LOWER]
succeed
Instruction ok
fail
no
S = jmp v
yes
Check according to rule [JMP]
succeed
Instruction ok
fail
no
S =halt [σ]
yes
Check according to rule [HALT]
succeed
Instruction ok
fail

FIG. 11B

$$
\begin{array}{rl}
\textit{(contexts)} & \kappa ::= \bullet \mid \theta \triangleright w \\
\textit{(pre-type)} & \tau ::= \mathrm{int} \mid \langle \sigma_1, \ldots, \sigma_n \rangle \mid {}^{A}[\Delta].\, \langle \kappa \rangle\, \Gamma \\
\textit{(types)} & \sigma ::= \tau_\theta \mid ns \\
\textit{(stack ty)} & \Sigma ::= \rho \mid nil \mid \sigma :: \Sigma \\
\textit{(var env)} & \Delta ::= {}_{\mathrm{o}} \mid \rho\Delta \mid \alpha\Delta \\
\textit{(type arg)} & \psi ::= \Sigma \mid w \\
\textit{(heap ty)} & \Psi ::= \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\} \\
\textit{(reg file ty)} & \Gamma ::= \{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \mathrm{sp} : \Sigma\} \\
\textit{(registers)} & r ::= r_1 \mid r_2 \mid \ldots \\
\textit{(word val)} & w ::= \alpha \mid l \mid i \mid ns \mid w \mid [\psi] \\
\textit{(small val)} & v ::= r \mid w \mid v\,[\psi] \\
\textit{(heap val)} & h ::= \langle w_1, \ldots, w_n \rangle \mid \mathrm{code}[\Delta]\, \langle \kappa \rangle\, \Gamma.I \\
\textit{(heaps)} & H ::= \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} \\
\textit{(reg files)} & R ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathrm{sp} \mapsto S\} \\
\textit{(stacks)} & S ::= nil \mid w :: S \\
\textit{(instr)} & i ::= \mathrm{add}\ r_d, r_s, v \mid \mathrm{ld}\ r_d, r_s(i) \mid \mathrm{st}\ r_d(i), r_s \mid \mathrm{mov}\ r_d, , v \\
 & \quad \mid \mathrm{bnz}\ r, v \mid \mathrm{salloc}\ i \mid \mathrm{sfree}\ i \mid \mathrm{sld}\ r_d, \mathrm{sp}(i) \\
 & \quad \mid \mathrm{sst}\ \mathrm{sp}(i), r_s \mid \mathrm{raise}\ \kappa \\
\textit{(instr seq)} & I ::= i; I \mid \mathrm{lower}\ w \mid \mathrm{jmp}\ v \mid \mathrm{halt}\ [\sigma] \\
\textit{(prog)} & P ::= (H, R, I)\kappa
\end{array}
$$

FIG. 12

$(H, R, I)_\kappa \mapsto P$ where	
if I =	then P =
add $r_d, r_s, v; I'$	$(H, R\{r_d \mapsto (i+i')\}, I')_\kappa$ where $R(r_s) = i$ and $\hat{R}(v) = i'$
ld $r_d, r_s(i); I'$	$(H, R\{r_d \mapsto wi\}, I')_\kappa$ where $R(r_s) = l$ and $H(l) = \langle w_0,...,w_{n-1}\rangle$ with $0 \le i < n$
mov $r_d, v; I'$	$(H, R\{r_d \mapsto R(v)\}, I')_\kappa$
bnz $r, v; I'$	$(H, R, I')_\kappa$ when $R(r) = 0$
bnz $r, v; I'$	$(H, R, I'' [\psi/\Delta])_\kappa$ when $R(r) = i$ where $i \neq 0$ and $\hat{R}(v) = l\,[\vec{\psi}]$ and $H(l) =$ code $[\Delta]\langle\kappa'\rangle\Gamma.I''$
st $r_d(i), r_s; I'$	$(H\{l \mapsto \langle w_0,...,w_{i-1}, R(r_s), w_{i+1},...,w_{n-1}\rangle\}, R, I')_\kappa$ where $R(r_d) = l$ and $H(l) = \langle w_0,...,w_{n-1}\rangle$ with $0 \le i < n$
salloc i, I'	$(H, R\{\text{sp} \mapsto \underbrace{ns::...::ns}_{i}::R(\text{sp})\}, I')_\kappa$
sfree i, I'	$(H, R\{\text{sp} \mapsto S\}, I')_\kappa$ where $R(\text{sp}) = w_1::...::w_i::S$
sld r_d, sp(i); I'	$(H, R\{r_d \mapsto w_i\}, I')_\kappa$ where $R(\text{sp}) = w_0::...::w_i::S$ and $i \ge 0$
sst sp(i), r_s; I'	$(H, R\{\text{sp} \mapsto w_0::...::w_{i-1}::R(r_s)::S\}, I')_\kappa$ where $R(\text{sp}) = w_0::...::w_i::S$ and $i \ge 0$
raise κ'; I'	$(H, R, I')_{\kappa'}$
lower w	$(H, R, I'[\vec{\psi}/\Delta])_{\kappa'}$ where $w = l[\vec{\psi}]$ and $H(l) =$ code $[\Delta]\langle\kappa'\rangle\Gamma.I'$
jmp v	$(H, R, I'[\vec{\psi}/\Delta])_{\kappa'}$ where $\hat{R}(v) = l[\vec{\psi}]$ and $H(l) =$ code $[\Delta]\langle\kappa'\rangle\Gamma.I'$

where $\hat{R}(v) = \begin{cases} R(r) & \text{when } v = r \\ w & \text{when } v = w \\ \hat{R}(v')[\psi] & \text{when } v = v'\,[\psi] \end{cases}$

FIG. 13

Judgement	Meaning
$\Delta \vdash \kappa$	κ is a valid context
$\Delta \vdash \tau$	τ is a valid pre-type
$\Delta \vdash \sigma$	σ is a valid type
$\Delta \vdash \Sigma$	Σ is a valid stack type
$\vdash \psi$	ψ is a valid heap type
$\Delta \vdash \Gamma$	Γ is a valid register file type
$\Delta \vdash \Gamma_1 \subseteq \Gamma_2$	Register file type Γ_1 weakens Γ_2
$\vdash H : \psi$	Heap H has type ψ
$\psi \vdash S : \Sigma$	Stack S has type Σ
$\psi \vdash R : \Gamma$	Register file R has type Γ
$\psi \vdash h : \sigma$	Heap value h has type σ
$\psi; \Delta \vdash w : \sigma$	Word value w has type σ
$\psi; \Delta; \Gamma \vdash v : \sigma$	Small value v has type σ
$\psi; \Delta; \Gamma; \kappa \vdash I$	I is a valid sequence of instructions
$\psi; \Gamma \vdash P$	P is a valid program

FIG. 14

$$\Delta \vdash \bullet \qquad \frac{\mathit{freevar}(w) \subseteq \Delta}{\Delta \vdash \theta \triangleright w} \qquad \Delta \vdash \mathit{int}$$

$$\frac{\Delta \vdash \sigma_i}{\Delta \vdash \langle \sigma_i, \ldots, \sigma_n \rangle} \qquad \frac{\Delta \vdash \kappa \quad \Delta \vdash \Gamma}{\Delta \vdash {}^{A}[\Delta].\langle \kappa \rangle \Gamma}$$

$$\frac{\Delta \vdash \tau}{\Delta \vdash \tau_\theta} \qquad \Delta \vdash \mathit{ns}$$

$$\frac{\rho \in \Delta}{\Delta \vdash \rho} \qquad \Delta \vdash \mathit{nil} \qquad \frac{\Delta \vdash \sigma \quad \Delta \vdash \Sigma}{\Delta \vdash \sigma :: \Sigma}$$

$$\frac{\Delta \vdash \sigma_i}{\vdash \{l_1{:}\sigma_1 \ldots l_n{:}\sigma_n\}} \qquad \frac{}{\Delta \vdash \{r_1{:}\sigma_1 \ldots r_n{:}\sigma_n, \mathrm{sp}{:}\ \Sigma\}}$$

$$\frac{\Delta \vdash \sigma_i \quad \Delta \vdash \Sigma \quad m \leq n}{\Delta \vdash \{r_1{:}\sigma_1 \ldots r_m{:}\sigma_m, \mathrm{sp}{:}\ \Sigma\} \subseteq \{r_1{:}\sigma_1 \ldots r_n{:}\sigma_n, \mathrm{sp}{:}\ \Sigma\}}$$

$$\frac{\vdash \Psi \quad \Psi = \{l_1{:}\sigma_1 \ldots l_n{:}\sigma_n\} \quad \Psi \vdash h_i : \sigma_i}{\vdash \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} : \Psi}$$

$$\frac{\Gamma = \{r_1{:}\sigma_1, \ldots r_n{:}\sigma_n, \mathrm{sp}{:}\ \Sigma\} \quad \Psi \vdash w_i : \sigma_i \quad \Psi \vdash S : \Sigma}{\Psi \vdash \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathrm{sp} \mapsto S\} : \Gamma}$$

$$\frac{\Psi \vdash w_i : \sigma_i}{\Psi \vdash \langle w_1, \ldots, w_n \rangle : \langle \sigma_1, \ldots, \sigma_n \rangle_\theta}$$

$$\frac{\Delta \vdash \kappa \quad \Delta \vdash \Gamma \quad \Psi; \Delta; \Gamma; \kappa \vdash I}{\Psi \vdash \mathrm{code}[\Delta]\langle \kappa \rangle \Gamma.I : ({}^{A}[\Delta].\langle \kappa \rangle \Gamma)_\theta}$$

$$\Psi; \Delta \vdash i : \mathrm{int}_\theta \qquad \Psi; \Delta \vdash \mathit{ns} : \mathit{ns} \qquad \frac{\Psi(l) = \sigma}{\Psi; \Delta \vdash i : \sigma}$$

$$\frac{\Delta \vdash \Sigma \quad \Psi; \Delta \vdash w : {}^{A}[\rho\Delta'].\langle \kappa \rangle \Gamma}{\Psi; \Delta \vdash w[\Sigma] : {}^{A}[\rho\Delta'].\langle \kappa[\Sigma/\rho] \rangle \Gamma[\Sigma/\rho]}$$

$$\frac{\Delta \vdash \Sigma \quad \Psi; \Delta \vdash w : {}^{A}[\alpha\Delta'].\langle \kappa \rangle \Gamma}{\Psi; \Delta \vdash w[w'] : {}^{A}[\Delta'].\langle \kappa[w'/\alpha] \rangle \Gamma[w'/\alpha]}$$

$$\frac{\Gamma(r) = \sigma}{\Psi; \Delta; \Gamma \vdash r : \sigma} \qquad \frac{\Psi; \Delta \vdash w : \sigma}{\Psi; \Delta; \Gamma \vdash w : \sigma}$$

$$\frac{\Delta \vdash \Sigma \quad \Psi; \Delta; \Gamma \vdash v : {}^{A}[\rho\Delta'].\langle \kappa \rangle \Gamma'}{\Psi; \Delta; \Gamma \vdash v[\Sigma] : {}^{A}[\Delta'].\langle \kappa[\Sigma/\rho] \rangle \Gamma'[\Sigma/\rho]}$$

$$\frac{\Delta \vdash \Sigma \quad \Psi; \Delta; \Gamma \vdash v : {}^{A}[\alpha\Delta'].\langle \kappa \rangle \Gamma'}{\Psi; \Delta; \Gamma \vdash v[w] : {}^{A}[\Delta'].\langle \kappa[w/\alpha] \rangle \Gamma'[w/\alpha]}$$

$$\frac{o \vdash \kappa \quad \vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; o; \Gamma; \kappa \vdash I}{\Psi : \Gamma \vdash (H, R, I)\kappa}$$

FIG. 15

$$\frac{SL(\kappa)=\theta \quad \Gamma(r_s)=\text{int}_{\theta_1} \quad \Psi;\Delta;\Gamma \vdash v : \text{int}_{\theta_2} \quad \Psi;\Delta;\Gamma\{r_d:\text{int}_{\theta \cup \theta_1 \cup \theta_2}\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{add } r_d, r_s, v; I}$$

$$\frac{SL(\kappa)=\theta \quad \Gamma(r_s)=\langle\sigma_1,\ldots,\sigma_n\rangle_{\theta_1} \quad \sigma_i=\tau_{\theta_2} \quad \Psi;\Delta;\Gamma\{r_d:\tau_{\theta \cup \theta_1 \cup \theta_2}\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{ld } r_d, r_s, (i); I}$$

$$\frac{SL(\kappa)=\theta \quad \Psi;\Gamma \vdash v : \tau_{\theta'} \quad \Psi;\Delta;\Gamma\{r_d:\tau_{\theta \cup \theta'}\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{mov} = r_d, v; I}$$

$$\frac{SL(\kappa)=\theta \quad \Gamma(r)=\text{int}_{\theta_1} \quad \Psi;\Delta;\Gamma \vdash v : ({}^{A}[o].\langle\kappa\rangle\Gamma')_{\theta_2} \quad \theta_1 \cup \theta_2 \subseteq \theta \quad \Delta \vdash \Gamma' \subseteq \Gamma \quad \Psi;\Delta;\Gamma;\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{bnz } r, v; I}$$

$$\frac{SL(\kappa)=\theta \quad \Gamma(r_d)=\langle\sigma_1,\ldots,\sigma_n\rangle_{\theta_1} \quad \sigma_i=\tau_{\theta'} \quad \Gamma(r_d)=\tau_{\theta_2} \quad \theta \cup \theta_1 \cup \theta_2 \subseteq \theta' \quad \Psi;\Delta;\Gamma;\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{st} = r_d(i), r_s; I}$$

$$\frac{\Gamma(\text{sp})=\Sigma \quad \Psi;\Delta;\Gamma\{\text{sp} : \overbrace{ns::\ldots::ns}^{i}::\Sigma\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{salloc } i;I}$$

$$\frac{\Gamma(\text{sp})=\sigma_1::\ldots::\sigma_i::\Sigma \quad \Psi;\Delta;\Gamma\{\text{sp} : \Sigma\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{sfree } i;I}$$

$$\frac{SL(\kappa)=\theta \quad \Gamma(\text{sp})=\sigma_0::\ldots::\sigma_i::\Sigma \quad \sigma_i=\tau_{\theta'} \quad \Psi;\Delta;\Gamma\{rd:\tau_{\theta \cup \theta'}\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{sld } r_d, \text{sp}(i); I}$$

$$\frac{SL(\kappa)=\theta \quad \Gamma(\text{sp})=\sigma_0::\ldots::\sigma_i::\Sigma \quad \Gamma(r_s)=\tau_{\theta'} \quad \Psi;\Delta;\Gamma\{\text{sp} : \sigma_0::\ldots::\sigma_{i-1}::\tau_{\theta \cup \theta'}::\Sigma\};\kappa \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{sst sp}(i) = r_s; I}$$

$$\frac{\kappa=\theta \triangleright w \quad \kappa'=\theta' \triangleright w' \quad \theta \subseteq \theta' \quad \Psi;\Delta \vdash w' : ({}^{A}[o].\langle\kappa\rangle\Gamma')_{\theta_1} \quad \Psi;\Delta;\Gamma;\kappa' \vdash I}{\Psi;\Delta;\Gamma;\kappa \vdash \text{raise } \kappa';I}$$

$$\frac{\kappa=\theta \triangleright w \quad \Psi=\Delta \vdash w : ({}^{A}[o].\langle\kappa'\rangle\Gamma')_{\theta_1} \quad \theta_1 \subseteq SL(\kappa') \quad \Delta \vdash \Gamma' \subseteq \Gamma}{\Psi;\Delta;\Gamma;\kappa \vdash \text{lower } u}$$

$$\frac{SL(\kappa)=\theta \quad \Psi=\Delta \vdash v : ({}^{A}[o].\langle\kappa\rangle\Gamma')_{\theta_1} \quad \theta_1 \subseteq \theta \quad \Delta \vdash \Gamma' \subseteq \Gamma}{\Psi;\Delta;\Gamma;\kappa \vdash \text{jmp } v}$$

$$\frac{\kappa=\bullet \quad \Delta \vdash \sigma \quad \Gamma(r_1)=\sigma}{\Psi;\Delta;\Gamma;\kappa \vdash \text{halt } [\sigma]}$$

FIG. 16

[TREconst] $$|\mathrm{i}| = \text{mov } r,i \,\|\, \mathrm{r}$$

[TREvar] $$|\mathrm{v}| = \text{mov } r,\ l_v;\ \text{ld } r',\ r(0) \,\|\, r'$$

[TREarg] $$|\mathrm{x}_i| = \text{sld } r,\ \text{sp}(i);\ \text{ld } r',\ r(0) \,\|\, r'$$

[TREadd] $$\frac{|\mathrm{E}| = \vec{i} \,\|\, r \quad |\mathrm{E}'| = \vec{i}' \,\|\, r' \quad \vec{i}' \text{ does not use } r}{|\mathrm{E}+\mathrm{E}'| = \vec{i};\ \vec{i}';\ \text{add } r'',\ r,\ r' \,\|\, r''}$$

FIG. 17A

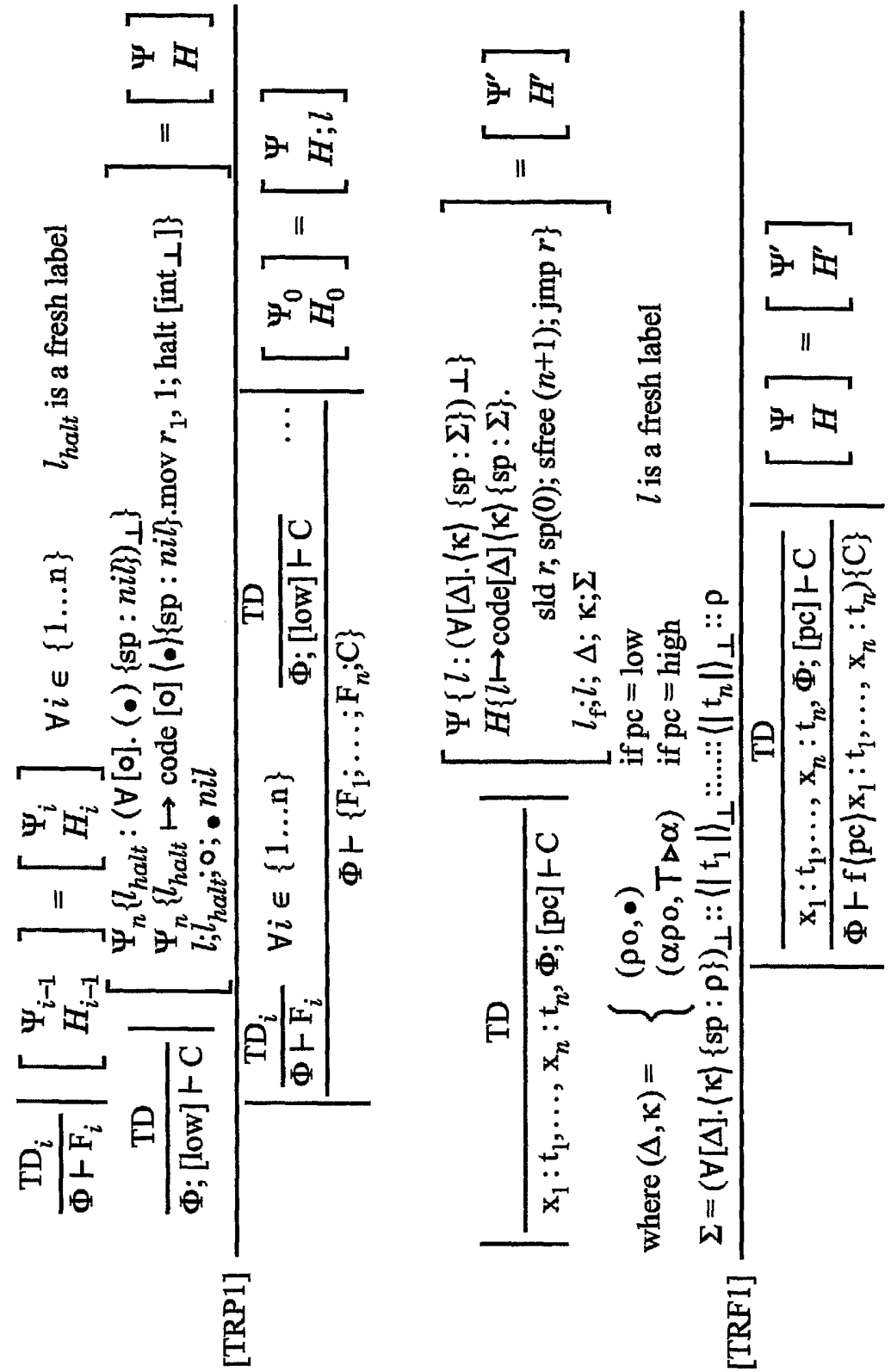
[TRP1]
[TRF1]
where (Δ, κ) =
if pc = low
if pc = high
l is a fresh label
l_{halt} is a fresh label
sld r, sp(0); sfree (n+1); jmp r'
l; l_{halt}; o; • nil

FIG. 17B

$$[\text{TRC1}]\quad \frac{|\mathrm{E}|=\vec{\tau}\,\|\,r}{\left|\frac{\Phi(\mathrm{V})=\text{high}}{\Phi;[\mathrm{pc}]\ \mathrm{V}:=\mathrm{E}}\right|\begin{bmatrix}\Psi\\ H\\ l;l';\Delta;\kappa;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi\{l:(\forall[\Delta].\langle\kappa\rangle\{\mathrm{sp}:\Sigma\})_{\perp}\}\\ H\{l\mapsto\mathrm{code}[\Delta]\langle\kappa\rangle\{\mathrm{sp}:\Sigma\}.\vec{\tau};\ \mathrm{mov}\ r',l_v;\ \mathrm{st}\ r'(0),r;\ \mathrm{jmp}\ l'[\Delta]\}\end{bmatrix}}$$

$$[\text{TRC2}]\quad \frac{|\mathrm{E}|=\vec{\tau}\,\|\,r}{\left|\frac{\Phi(\mathrm{V})=\text{low}\ \ \Phi\vdash\mathrm{E}:\text{low}}{\Phi;[\text{low}]\vdash\mathrm{V}:=\mathrm{E}}\right|\begin{bmatrix}\Psi\\ H\\ l;l';\Delta;\bullet;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi\{l:(\forall[\Delta].\langle\bullet\rangle\{\mathrm{sp}:\Sigma\})_{\perp}\}\\ H\{l\mapsto\mathrm{code}[\Delta]\langle\bullet\rangle\{\mathrm{sp}:\Sigma\}.\vec{\tau};\ \mathrm{mov}\ r',l_1;\ \mathrm{st}\ r'(0),r;\ \mathrm{jmp}\ l'[\Delta]\}\end{bmatrix}}$$

$$[\text{TRC3}]\quad \frac{|\mathrm{E}|=\vec{\tau}\,\|\,r\qquad l_1,l_2\text{ are fresh labels}\qquad \left|\frac{\mathrm{TD}_1}{\Phi;[\mathrm{pc}]\vdash\mathrm{C}_1}\right|\begin{bmatrix}\Psi\\ H\\ l_1;l';\Delta;\kappa;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_1\\ H_1\end{bmatrix}\qquad \left|\frac{\mathrm{TD}_2}{\Phi;[\mathrm{pc}]\vdash\mathrm{C}_2}\right|\begin{bmatrix}\Psi_1\\ H_1\\ l_2;l';\Delta;\kappa;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_2\\ H_2\end{bmatrix}}{\left|\frac{\Phi\vdash\mathrm{E}:\mathrm{pc}\ \frac{\mathrm{TD}_1}{\Phi;[\mathrm{pc}]\vdash\mathrm{C}_1}\ \frac{\mathrm{TD}_2}{\Phi;[\mathrm{pc}]\vdash\mathrm{C}_2}}{\Phi;[\mathrm{pc}]\vdash\text{if E then }\mathrm{C}_1\text{ else }\mathrm{C}_2}\right|\begin{bmatrix}\Psi\\ H\\ l;l';\Delta;\kappa;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_2\{l:(\forall[\Delta].\langle\kappa\rangle\{\mathrm{sp}:\Sigma\})_{\perp}\}\\ H_2\{l\mapsto\mathrm{code}[\Delta]\langle\kappa\rangle\{\mathrm{sp}:\Sigma\}.\vec{\tau};\ \mathrm{bnz}\ r,l_1[\Delta];\ \mathrm{jmp}\ l_2[\Delta]\}\end{bmatrix}}$$

$$[\text{TRC4}]\quad \frac{\left|\frac{\mathrm{TD}_1}{\Phi;[\text{high}]\vdash\mathrm{C}}\right|\begin{bmatrix}\Psi\{l_1:(\forall[\Delta].\langle\top\triangleright l'[\Delta]\rangle\{\mathrm{sp}:\Sigma\})_{\perp}\}\\ H\{l_1\mapsto\mathrm{code}[\Delta]\langle\top\triangleright l'[\Delta]\rangle\{\mathrm{sp}:\Sigma\}.\text{lower }l'[\Delta]\}\\ l_0;l_1;\Delta;\top\triangleright l'[\Delta];\Sigma\end{bmatrix}=\begin{bmatrix}\Psi'\\ H'\end{bmatrix}\qquad l_0;l_1\text{ are fresh labels}}{\left|\frac{\mathrm{TD}}{\frac{\Phi;[\text{high}]\vdash\mathrm{C}}{\Phi;[\text{low}]\vdash\mathrm{C}}}\right|\begin{bmatrix}\Psi\\ H\\ l;l';\Delta;\bullet;\Sigma\end{bmatrix}=\begin{bmatrix}\Psi\{l:(\forall[\Delta].\langle\bullet\rangle\{\mathrm{sp}:\Sigma\})_{\perp}\}\\ H'\{l\mapsto\mathrm{code}[\Delta]\langle\bullet\rangle\{\mathrm{sp}:\Sigma\}.\text{raise }\top\triangleright l'[\Delta];\ \mathrm{jmp}\ l_0[\Delta]\}\end{bmatrix}}$$

FIG. 17C1

To FIG. 17C2 ↓

From FIG. 17C1 ↓

[TRC5]
$$\frac{\left|\frac{TD_2}{\Phi;\,[pc]\vdash C_2}\right|\begin{bmatrix}\Psi\\ H\\ l_1;l';\,\Delta;\,\kappa;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_1\\ H_1\end{bmatrix}\qquad \left|\frac{TD_1}{\Phi;\,[pc]\vdash C_1}\right|\begin{bmatrix}\Psi_1\\ H_1\\ l;l_1;\,\Delta;\,\kappa;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_2\\ H_2\end{bmatrix}}{\left|\frac{\dfrac{TD_1}{\Phi;\,[pc]\vdash C_1}\ \dfrac{TD_2}{\Phi;\,[pc]\vdash C_2}}{\Phi;\,[pc]\vdash C_1\,;\,C_2}\right|\begin{bmatrix}\Psi\\ H\\ l;\,l';\,\Delta;\,\kappa\,;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi_2\\ H_2\end{bmatrix}}$$

[TRC6]
$$\frac{|E|=\vec{\imath}\,\|\,r\ \left|\frac{TD}{\Phi;\,[pc]\vdash C}\right|\begin{bmatrix}\Psi\{l:(\forall[\Delta].\langle\kappa\rangle\{sp:\Sigma\})_\perp\}\\ H\{l\mapsto \text{code}[\Delta]\langle\kappa\rangle\{sp:\Sigma\}.\text{jmp } l[\Delta]\}\\ l_1;\,l;\,\Delta;\,\kappa\,;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi'\\ H'\end{bmatrix}}{\left|\frac{\Phi\vdash E:pc\quad \dfrac{TD_1}{\Phi;\,[pc]\vdash C}}{\Phi;\,[pc]\vdash \text{while E do C}}\right|\begin{bmatrix}\Psi\\ H\\ l;\,l';\,\Delta;\,\kappa\,;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi'\{l:(\forall[\Delta].\langle\kappa\rangle\{sp:\Sigma\})_\perp\}\\ H'\{l\mapsto\text{code}[\Delta]\langle\kappa\rangle\{sp:\Sigma\}.\vec{\imath};\ \text{bnz } r,\,l_1[\Delta];\ \text{jmp } l'[\Delta]\}\end{bmatrix}}$$

$|V_i|=i_i\|r_i$ where $\vec{\imath}_i$ does not use r_j if $j<i\ \ \forall i\in\{1\ldots n\}$

$\Psi'=\Psi\{l:(\forall[\Delta].\langle\kappa\rangle\{sp:\Sigma\})_\perp\}$

$H'=\Psi\{l\mapsto\text{code}[\Delta]\langle\kappa\rangle\{sp:\Sigma\}.\vec{\imath}_1;\ldots\vec{\imath}_n;\ \text{salloc}(n+1);\ \text{sst sp}(0),l'[\vec{\psi}];\quad \text{sst sp}(1),r_1;\ldots\ \text{sst sp}(n),r_n;\ \text{jmp } l_f[\vec{\psi}]\}$ where $\vec{\psi}=\begin{cases}\Sigma & \text{if }\kappa=\bullet\\ w\Sigma & \text{if }\kappa=\top\triangleright w\end{cases}$

[TRC7]
$$\frac{\left|\frac{\Phi(f)=\langle pc\rangle(t_1,\ldots,t_n)\to\text{void}\quad \dfrac{TD_i}{\Phi\vdash V_i:t_i}\ \forall_i\in\{1\ldots n\}}{\Phi;\,[pc]\vdash f(V_1,\ldots,V_n)}\right|\begin{bmatrix}\Psi\\ H\\ l;\,l';\,\Delta;\,\kappa\,;\,\Sigma\end{bmatrix}=\begin{bmatrix}\Psi'\\ H'\end{bmatrix}}{}$$

FIG. 17C2

```
int double(x:int) {x=x*2;}
...
if h<10 then double (h);
double (1); ...
```

FIG. 18

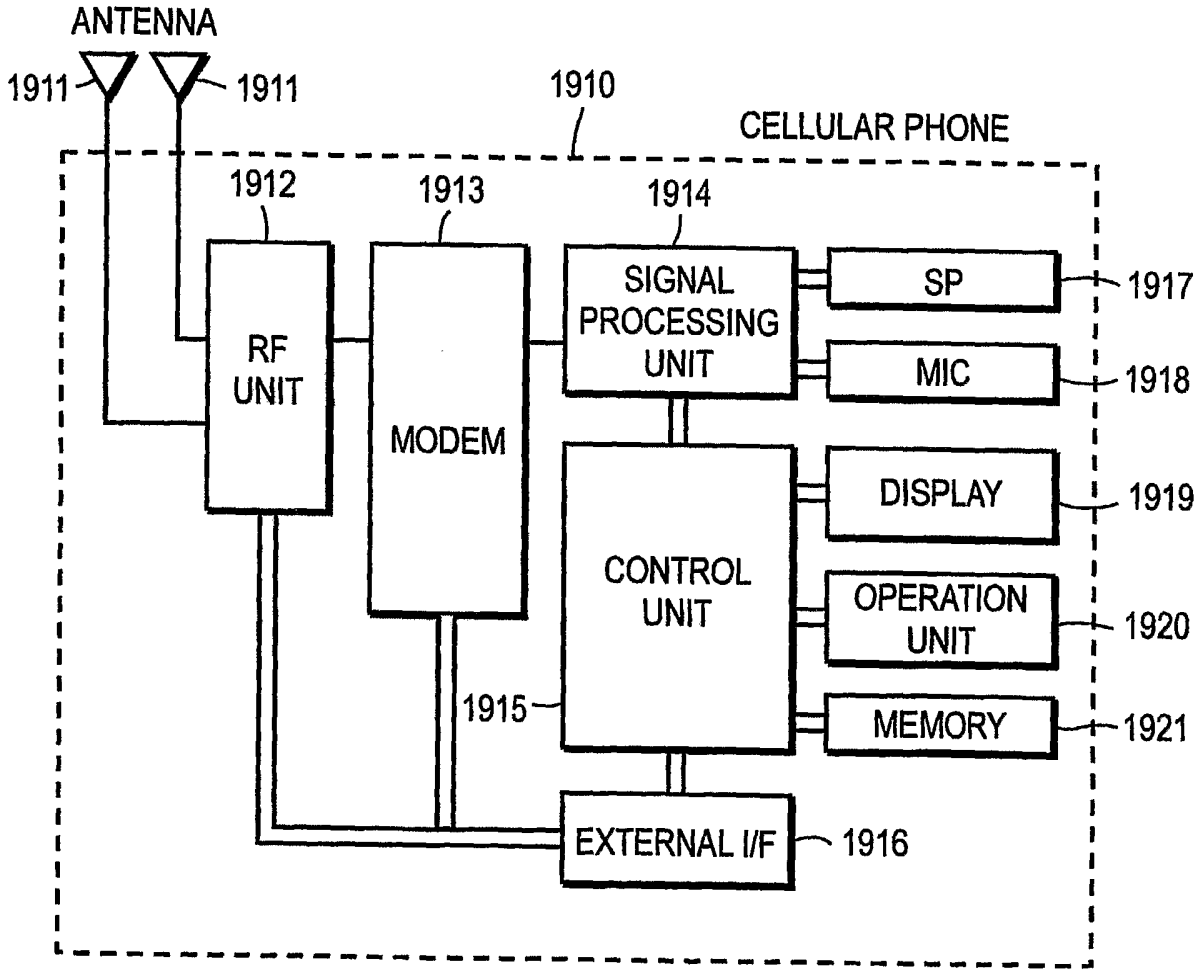

FIG. 19

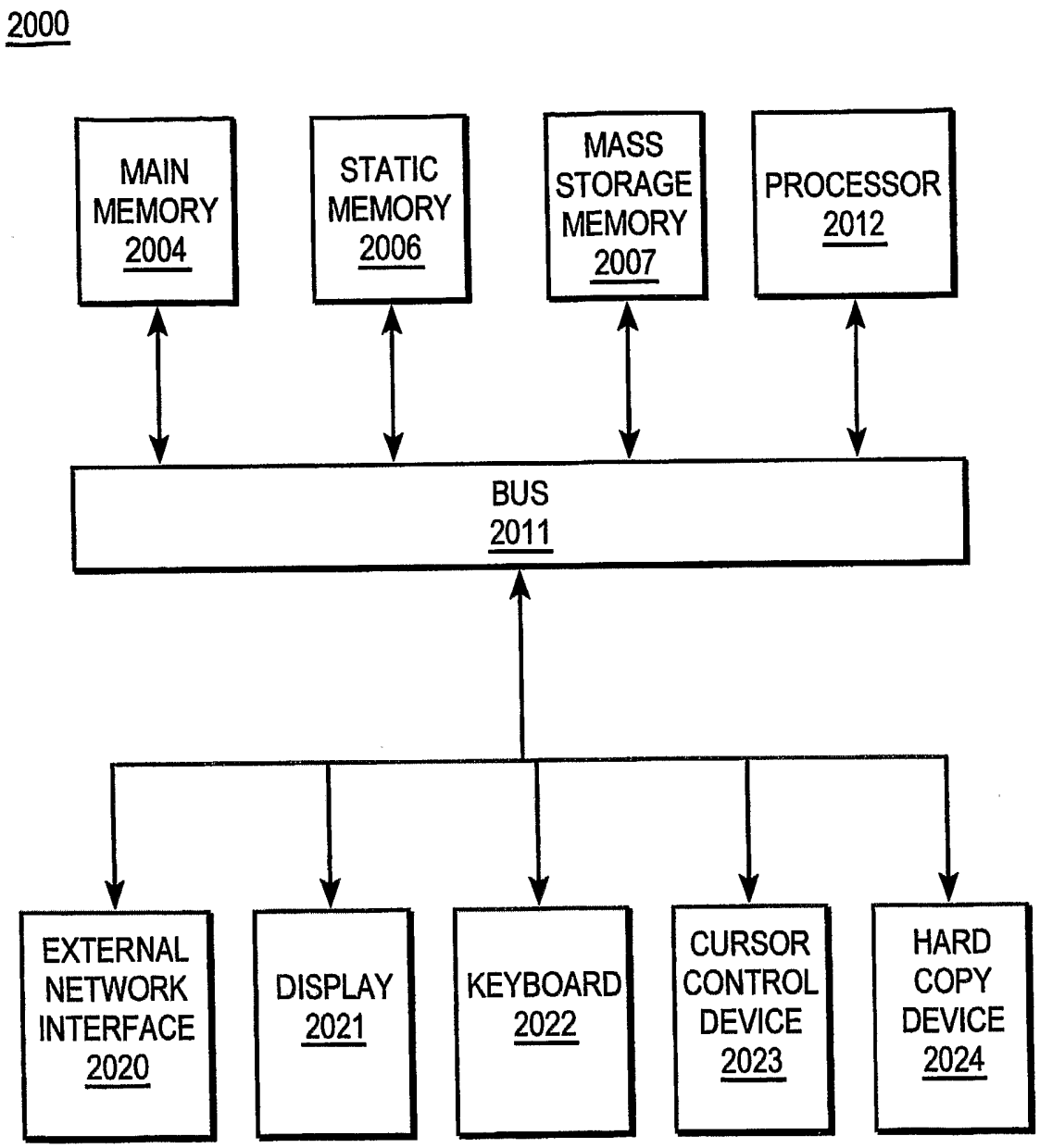

FIG. 20

SUBSTITUTE SHEET (RULE 26)