EXHIBIT A

EXHIBIT B-2
Second Supplemental Infringement Contentions for the '205 Patent

NOTE: The infringement evidence cited below is exemplary and not exhaustive. The cited examples are taken from Android 2.2,
2.3, and Google's Android websites. ~~Oracle's infringement contentions apply to all versions of Android having similar or nearly identical code or documentation, including past and expected future releases. Although Oracle's investigation is ongoing, the~~The '205 patent is infringed by all versions of Android ~~from Oct. 21, 2008 to the present~~through 2.3 ("Gingerbread"), including Android 1.1, 1.5 ("Cupcake"), 1.6 ("Donut"), 2.0/2.1 ("Éclair"), 2.2 ("Froyo"), and 2.3 ("Gingerbread").

The cited source code examples ~~are~~were taken from http://android.git.kernel.org/. The citations are shortened and mirror the file paths shown in http://android.git.kernel.org/. For example, "dalvik\vm\native\InternalNative.c" maps to "[platform/dalvik.git] / vm / native / InternalNative.c" (accessible at http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/native/InternalNative.c). Google has apparently made modifications to certain source code files since Oracle's Preliminary Infringement Contentions were served on December 2, 2010. As such, file paths may refer to earlier versions of Android than what is immediately available at http://android.git.kernel.org/.~~It~~ In addition, since Oracle's Supplemental Infringement Contentions were served on April 1, 2011, it appears that ~~the~~Google has taken down http://android.git.kernel.org and now makes Android ~~git~~ source code ~~repository (accessible~~available through ~~http://android.git.kernel.org/) was created on or around Oct. 21, 2008. As such, the list of infringing Android versions may be expanded based on what Oracle learns about earlier Android versions.~~its own servers. (See http://source.android.com/source/downloading.html.)

Oracle has determined that Android devices execute much of the code cited below every time the devices start up. Other cited code is invoked when a developer runs the Android Compatibility Test Suite (CTS), which Google requires manufacturers to execute to certify devices as Android-compatible.[1] The mobile device emulator that Google includes with the Android SDK[2] supports Oracle's conclusion. The emulator displays log messages to inform developers of what is running on the virtual

[1] http://source.android.com/compatibility/android-2.2-cdd.pdf at 10 ("To be considered compatible with Android 2.2, device implementations . . . MUST pass the most recent version of the Android Compatibility Test Suite (CTS) available at the time of the device implementation's software is completed.").

[2] See http://developer.android.com/guide/developing/devices/emulator.html ("The Android SDK includes a virtual mobile device emulator that runs on your computer. The emulator lets you prototype, develop, and test Android applications without using a physical device. The Android emulator mimics all of the hardware and software features of a typical mobile device, except that it cannot place actual phone calls.").

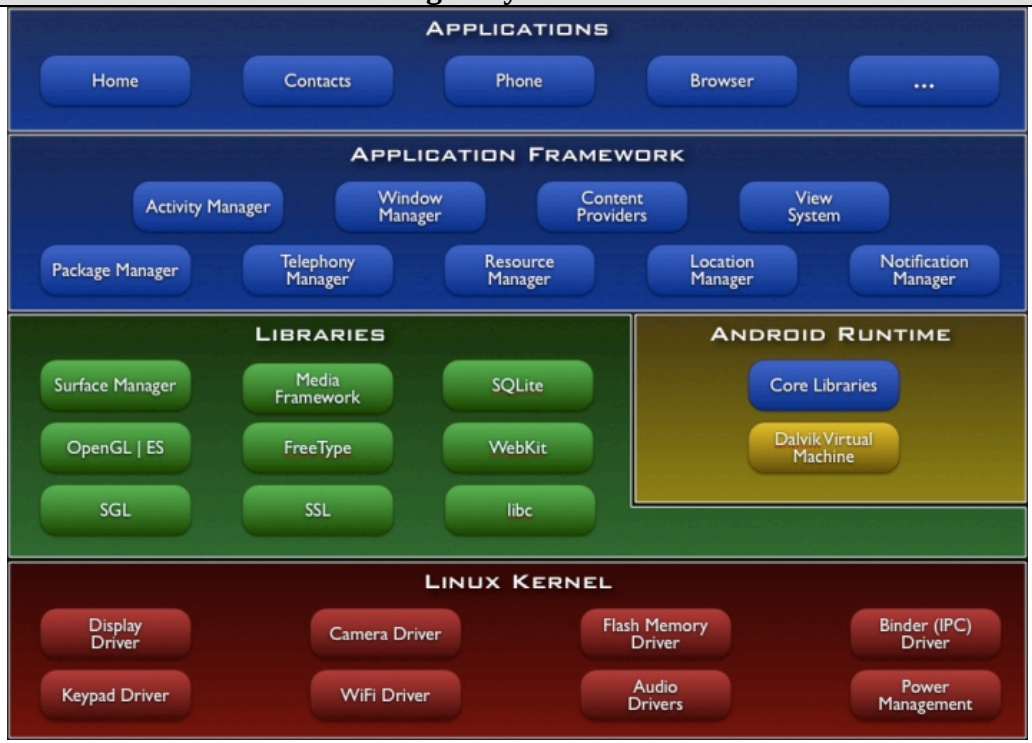
device. If the developer includes a logging command in part of a program, the emulator will output a log entry every time that part of the program is executed. A developer might use this feature, for example, to test whether an application starts to execute a particular section of code before failing. By adding logging commands to key portions of the Android source code cited below, building an Android system image, and loading it into Google's emulator, Oracle determined that many of these code portions are executed even before a user can interact with a device. Thus, Android-compatible devices, when used as Google intends, execute infringing code.

The asserted claims include method claims. Anyone who uses a device running Android directly infringes the method claims. This includes Google and its downstream licensees, including device manufacturers, carriers, application developers, and end users. Google induces and contributes to infringement of all asserted claims by distributing Android code with the intention that it will be executed on mobile devices. Much of the code cited below is executed not only as applications run, but every time a device running Android starts up. Thus Android is not a staple article suitable for substantial non-infringing use.

The '205 Patent	**Infringed By**
1. In a computer system, a method for increasing the execution speed of virtual machine instructions at runtime, the method comprising:	Android uses the Dalvik virtual machine to execute virtual machine bytecode instructions at runtime. The Dalvik virtual machine performs and runs code resulting from certain optimizations to increase the execution speed of virtual machine instructions at runtime. *See, e.g.,* Android Glossary Definition for "Dalvik," available at http://developer.android.com/guide/appendix/glossary.html: Dalvik The Android platform's virtual machine. The Dalvik VM is an interpreter-only virtual machine that executes files in the Dalvik Executable (.dex) format, a format that is optimized for efficient storage and memory-mappable execution. The virtual machine is register-based, and it can run classes compiled by a Java language compiler that have been transformed into its native format using the included "dx" tool. The VM runs on top of Posix-compliant operating systems, which it relies on for underlying functionality (such as threading and low level memory management). The Dalvik core class library is intended to provide a familiar development base for those used to programming with Java Standard Edition, but it is geared specifically to the needs of a small mobile device.

The ’205 Patent	Infringed By
	Android Basics, entitled “What is Android?,” available at http://developer.android.com/guide/basics/what-is-android.html. **What is Android?** Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language. **Features** • Application framework enabling reuse and replacement of components • Dalvik virtual machine optimized for mobile devices • Integrated browser based on the open source WebKit engine • Optimized graphics powered by a custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional) • SQLite for structured data storage • Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF) • GSM Telephony (hardware dependent) • Bluetooth, EDGE, 3G, and WiFi (hardware dependent) • Camera, GPS, compass, and accelerometer (hardware dependent) • Rich development environment including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE **Android Architecture** The following diagram shows the major components of the Android operating system. Each section is described in more detail below.

The ’205 Patent	Infringed By



Applications

Android will ship with a set of core applications including an email client, SMS program, calendar, maps, browser, contacts, and others. All applications are written using the Java programming language.

…

Android Runtime

Android includes a set of core libraries that provides most of the functionality available

The ’205 Patent	Infringed By
	in the core libraries of the Java programming language. Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. Dalvik has been written so that a device can run multiple VMs efficiently. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory footprint. The VM is register-based, and runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool. The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management. Android uses the dexopt tool, which increases the execution speed of virtual machine instructions at runtime: *See, e.g.*, dalvik\docs\dexopt.html; *see also*, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html: **Dalvik Optimization and Verification With *dexopt*** The Dalvik virtual machine was designed specifically for the Android mobile platform. The target systems have little RAM, store data on slow internal flash memory, and generally have the performance characteristics of decade-old desktop systems. They also run Linux, which provides virtual memory, processes and threads, and UID-based security mechanisms. The features and limitations caused us to focus on certain goals: • Class data, notably bytecode, must be shared between multiple processes to minimize total system memory usage. • The overhead in launching a new app must be minimized to keep the device responsive. • Storing class data in individual files results in a lot of redundancy, especially with respect to strings. To conserve disk space we need to factor this out. • Parsing class data fields adds unnecessary overhead during class loading. Accessing data

The ’205 Patent	Infringed By
	values (e.g. integers and strings) directly as C types is better. • Bytecode verification is necessary, but slow, so we want to verify as much as possible outside app execution. • Bytecode optimization (quickened instructions, method pruning) is important for speed and battery life. • For security reasons, processes may not edit shared code. The typical VM implementation uncompresses individual classes from a compressed archive and stores them on the heap. This implies a separate copy of each class in every process, and slows application startup because the code must be uncompressed (or at least read off disk in many small pieces). On the other hand, having the bytecode on the local heap makes it easy to rewrite instructions on first use, facilitating a number of different optimizations. The goals led us to make some fundamental decisions: • Multiple classes are aggregated into a single "DEX" file. • DEX files are mapped read-only and shared between processes. • Byte ordering and word alignment are adjusted to suit the local system. • Bytecode verification is mandatory for all classes, but we want to "pre-verify" whatever we can. • Optimizations that require rewriting bytecode must be done ahead of time. • The consequences of these decisions are explained in the following sections. …. **dexopt** We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in. The solution is to invoke a program called dexopt, which is really just a back door into the VM.

The ’205 Patent	Infringed By
	It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources. It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once. …. **Optimization** Virtual machine interpreters typically perform certain optimizations the first time a piece of code is used. Constant pool references are replaced with pointers to internal data structures, operations that always succeed or always work a certain way are replaced with simpler forms. Some of these require information only available at runtime, others can be inferred statically when certain assumptions are made. The Dalvik optimizer does the following: • For virtual method calls, replace the method index with a vtable index. • For instance field get/put, replace the field index with a byte offset. Also, merge the boolean / byte / char / short variants into a single 32-bit form (less code in the interpreter means more room in the CPU I-cache). • Replace a handful of high-volume calls, like String.length(), with "inline" replacements. This skips the usual method call overhead, directly switching from the interpreter to a native implementation. • Prune empty methods. The simplest example is Object.<init>, which does nothing, but must be called whenever any object is allocated. The instruction is replaced with a new version that acts as a no-op unless a debugger is attached. • Append pre-computed data. For example, the VM wants to have a hash table for lookups on class name. Instead of computing this when the DEX file is loaded, we can compute it now, saving heap space and computation time in every VM where the DEX is loaded.

The ’205 Patent	Infringed By
	All of the instruction modifications involve replacing the opcode with one not defined by the Dalvik specification. This allows us to freely mix optimized and unoptimized instructions. The set of optimized instructions, and their exact representation, is tied closely to the VM version. Most of the optimizations are obvious "wins". The use of raw indices and offsets not only allows us to execute more quickly, we can also skip the initial symbolic resolution. Pre-computation eats up disk space, and so must be done in moderation. There are a couple of potential sources of trouble with these optimizations. First, vtable indices and byte offsets are subject to change if the VM is updated. Second, if a superclass is in a different DEX, and that other DEX is updated, we need to ensure that our optimized indices and offsets are updated as well. A similar but more subtle problem emerges when user-defined class loaders are employed: the class we actually call may not be the one we expected to call. These problems are addressed with dependency lists and some limitations on what can be optimized. *See also, e.g.*, dalvik\docs\ embedded-vm-control.html#verifier (“The system tries to pre-verify all classes in a DEX file to reduce class load overhead, and performs a series of optimizations to improve runtime performance. Both of these are done by the dexopt command, either in the build system or by the installer. On a development device, dexopt may be run the first time a DEX file is used and whenever it or one of its dependencies is updated ("just-in-time" optimization and verification).”).

receiving a first virtual machine instruction;	Android’s dexopt tool receives a first virtual machine instruction.
	
	See, e.g.,
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l1486
	`1486 /*`
	`1487 * Run through all classes that were successfully loaded from this DEX`
	`1488 * file and optimize their code sections.`
	`1489 */`
	`1490 static void optimizeLoadedClasses(DexFile* pDexFile)`
	`1491 {`
	`1492 u4 count = pDexFile->pHeader->classDefsSize;`

The ’205 Patent	Infringed By

	1493 u4 idx;
	1494 InlineSub* inlineSubs = NULL;
	1495
	1496 LOGV("DexOpt: +++ optimizing up to %d classes\n", count);
	1497 assert(gDvm.dexOptMode != OPTIMIZE_MODE_NONE);
	1498
	1499 inlineSubs = createInlineSubsTable();
	1500
	1501 for (idx = 0; idx < count; idx++) {
	1502 const DexClassDef* pClassDef;
	1503 const char* classDescriptor;
	1504 ClassObject* clazz;
	1505
	1506 pClassDef = dexGetClassDef(pDexFile, idx);
	1507 classDescriptor = dexStringByTypeIdx(pDexFile, pClassDef->classIdx);
	1508
	1509 /* all classes are loaded into the bootstrap class loader */
	1510 clazz = dvmLookupClass(classDescriptor, NULL, false);
	1511 if (clazz != NULL) {
	1512 if ((pClassDef->accessFlags & CLASS_ISPREVERIFIED) == 0 &&
	1513 gDvm.dexOptMode == OPTIMIZE_MODE_VERIFIED)
	1514 {
	1515 LOGV("DexOpt: not optimizing '%s': not verified\n",
	1516 classDescriptor);
	1517 } else if (clazz->pDvmDex->pDexFile != pDexFile) {
	1518 /* shouldn't be here -- verifier should have caught */
	1519 LOGD("DexOpt: not optimizing '%s': multiple definitions\n",
	1520 classDescriptor);
	1521 } else {
	1522 optimizeClass(clazz, inlineSubs);
	1523
	1524 /* set the flag whether or not we actually did anything */
	1525 ((DexClassDef*)pClassDef)->accessFlags |=
	1526 CLASS_ISOPTIMIZED;
	1527 }
	1528 } else {
	1529 LOGV("DexOpt: not optimizing unavailable class '%s'\n",
	1530 classDescriptor);
	1531 }

The '205 Patent	Infringed By

	1532 }
	1533
	1534 free(inlineSubs);
	1535 }
	
	At 1501..1522..1532, Android calls `optimizeClass()` for each class in DEX file, passing in a table of inline substitutions, inlineSubs. *See also, e.g.*, dalvik\vm\analysis\DexPrepare.c and dalvik\vm\analysis\Optimize.c (*e.g.*, dvmCreateInlineSubsTable routines).
	
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l1537
	1537 /*
	1538 * Optimize the specified class.
	1539 */
	1540 static void optimizeClass(ClassObject* clazz, const InlineSub*
	inlineSubs)
	1541 {
	1542 int i;
	1543
	1544 for (i = 0; i < clazz->directMethodCount; i++) {
	1545 if (!optimizeMethod(&clazz->directMethods[i], inlineSubs))
	1546 goto fail;
	1547 }
	1548 for (i = 0; i < clazz->virtualMethodCount; i++) {
	1549 if (!optimizeMethod(&clazz->virtualMethods[i], inlineSubs))
	1550 goto fail;
	1551 }
	1552
	1553 return;
	1554
	1555 fail:
	1556 LOGV("DexOpt: ceasing optimization attempts on %s\n", clazz-
	>descriptor);
	1557 }
	
	At 1544-1545..1547 and 1548-1549..1551 Android calls `optimizeMethod()` on each direct or virtual method in the incoming DEX class. *See also, e.g.*, dalvik\vm\analysis\Optimize.c.

The ’205 Patent	Infringed By
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l1559

```
1559 /*
1560 * Optimize instructions in a method.
1561 *
1562 * Returns "true" if all went well, "false" if we bailed out early when
1563 * something failed.
1564 */
1565 static bool optimizeMethod(Method* method, const InlineSub* inlineSubs)
1566 {
1567 u4 insnsSize;
1568 u2* insns;
1569 u2 inst;
1570
1571 if (dvmIsNativeMethod(method) || dvmIsAbstractMethod(method))
1572 return true;
1573
1574 insns = (u2*) method->insns;
1575 assert(insns != NULL);
1576 insnsSize = dvmGetMethodInsnsSize(method);
1577
1578 while (insnsSize > 0) {
1579 int width;
1580
1581 inst = *insns & 0xff;
1582
1583 switch (inst) {
...
1645 case OP_INVOKE_DIRECT_RANGE:
1646 rewriteExecuteInlineRange(method, insns, METHOD_DIRECT, inlineSubs);
1647 break;
...
1652 case OP_INVOKE_STATIC_RANGE:
1653 rewriteExecuteInlineRange(method, insns, METHOD_STATIC, inlineSubs);
1654 break;
...
1656 default:
1657 // ignore this instruction
1658 ;
1659 }
```

The '205 Patent	Infringed By
	... 1674 insns += width; 1675 insnsSize -= width; 1676 } 1677 1678 assert(insnsSize == 0); 1679 return true; 1680 } 1578..1645-1647..1652-1654..1676 Android calls `rewriteExecuteInlineRange()` for each OP_INVOKE_DIRECT_RANGE or OP_INVOKE_STATIC_RANGE virtual machine instruction. *See also, e.g.*, dalvik\vm\analysis\Optimize.c.
generating, at runtime, a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction; and	Android generates at runtime a new virtual machine instruction that represents or references one or more native instructions that can be executed instead of said first virtual machine instruction. As described above, Android includes a utility called dexopt: See, e.g., dalvik\docs\dexopt.html; see also, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=docs/dexopt.html: **dexopt** We want to verify and optimize all of the classes in the DEX file. The easiest and safest way to do this is to load all of the classes into the VM and run through them. Anything that fails to load is simply not verified or optimized. Unfortunately, this can cause allocation of some resources that are difficult to release (e.g. loading of native shared libraries), so we don't want to do it in the same virtual machine that we're running applications in. The solution is to invoke a program called dexopt, which is really just a back door into the VM. It performs an abbreviated VM initialization, loads zero or more DEX files from the bootstrap class path, and then sets about verifying and optimizing whatever it can from the target DEX. On completion, the process exits, freeing all resources. It is possible for multiple VMs to want the same DEX file at the same time. File locking is used to ensure that dexopt is only run once.

The ’205 Patent	Infringed By

	….
	
	See also, e.g., dalvik\docs\ embedded-vm-control.html#verifier (“The system tries to pre-verify all classes in a DEX file to reduce class load overhead, and performs a series of optimizations to improve runtime performance. Both of these are done by the dexopt command, either in the build system or by the installer. On a development device, dexopt may be run the first time a DEX file is used and whenever it or one of its dependencies is updated ("just-in-time" optimization and verification).”).
	
	See, e.g.,
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l2345
	`2345 /*`
	`2346 * See if the method being called can be rewritten as an inline operation.`
	`2347 * Works for invoke-virtual/range, invoke-direct/range, and invoke-static/range.`
	`2348 *`
	`2349 * Returns "true" if we replace it.`
	`2350 */`
	`2351 static bool rewriteExecuteInlineRange(Method* method, u2* insns,`
	`2352 MethodType methodType, const InlineSub* inlineSubs)`
	`2353 {`
	`2354 ClassObject* clazz = method->clazz;`
	`2355 Method* calledMethod;`
	`2356 u2 methodIdx = insns[1];`
	`2357`
	`2358 calledMethod = dvmOptResolveMethod(clazz, methodIdx, methodType, NULL);`
	`2359 if (calledMethod == NULL) {`
	`2360 LOGV("+++ DexOpt inline/range: can't find %d\n", methodIdx);`
	`2361 return false;`
	`2362 }`
	`2363`
	`2364 while (inlineSubs->method != NULL) {`
	`2365 if (inlineSubs->method == calledMethod) {`
	`2366 assert((insns[0] & 0xff) == OP_INVOKE_DIRECT_RANGE ||`
	`2367 (insns[0] & 0xff) == OP_INVOKE_STATIC_RANGE ||`
	`2368 (insns[0] & 0xff) == OP_INVOKE_VIRTUAL_RANGE);`

The ’205 Patent	Infringed By

```
2369 insns[0] = (insns[0] & 0xff00) | (u2) OP_EXECUTE_INLINE_RANGE;
2370 insns[1] = (u2) inlineSubs->inlineIdx;
2371
2372 //LOGI("DexOpt: execute-inline/range %s.%s --> %s.%s\n",
2373 // method->clazz->descriptor, method->name,
2374 // calledMethod->clazz->descriptor, calledMethod->name);
2375 return true;
2376 }
2377
2378 inlineSubs++;
2379 }
2380
2381 return false;
2382 }
```

at 2369-70 Android generates a new instruction, with OP_EXECUTE_INLINE_RANGE as the new opcode. See also, e.g., dalvik\vm\analysis\Optimize.c.

at 2370 Android stores inlineSubs->inlineIdx (the index of the native code in the ~~inlineSubs table~~`gDvmInlineOpsTable`) as the instruction data to reference the native code.

Android passes in `const InlineSub* inlineSubs`, which is constructed at line 1499 by calling `createInlineSubsTable()`, which is:

http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l1411

```
1411 /*
1412 * Create a table of inline substitutions.
1413 *
1414 * TODO: this is currently just a linear array. We will want to put this
1415 * into a hash table as the list size increases.
1416 */
1417 static InlineSub* createInlineSubsTable(void)
1418 {
1419 const InlineOperation* ops = dvmGetInlineOpsTable();
1420 const int count = dvmGetInlineOpsTableLength();
1421 InlineSub* table;
1422 Method* method;
```

The ’205 Patent	Infringed By

	1423 ClassObject* clazz;
	1424 int i, tableIndex;
	1425
	1426 /*
	1427 * Allocate for optimism: one slot per entry, plus an end-of-list marker.
	1428 */
	1429 table = malloc(sizeof(InlineSub) * (count+1));
	1430
	1431 tableIndex = 0;
	1432 for (i = 0; i < count; i++) {
	1433 clazz = dvmFindClassNoInit(ops[i].classDescriptor, NULL);
	1434 if (clazz == NULL) {
	1435 LOGV("DexOpt: can't inline for class '%s': not found\n",
	1436 ops[i].classDescriptor);
	1437 dvmClearOptException(dvmThreadSelf());
	1438 } else {
	1439 /*
	1440 * Method could be virtual or direct. Try both. Don't use
	1441 * the "hier" versions.
	1442 */
	1443 method = dvmFindDirectMethodByDescriptor(clazz, ops[i].methodName,
	1444 ops[i].methodSignature);
	1445 if (method == NULL)
	1446 method = dvmFindVirtualMethodByDescriptor(clazz, ops[i].methodName,
	1447 ops[i].methodSignature);
	1448 if (method == NULL) {
	1449 LOGW("DexOpt: can't inline %s.%s %s: method not found\n",
	1450 ops[i].classDescriptor, ops[i].methodName,
	1451 ops[i].methodSignature);
	1452 } else {
	1453 if (!dvmIsFinalClass(clazz) && !dvmIsFinalMethod(method)) {
	1454 LOGW("DexOpt: WARNING: inline op on non-final class/method "
	1455 "%s.%s\n",
	1456 clazz->descriptor, method->name);
	1457 /* fail? */
	1458 }
	1459 if (dvmIsSynchronizedMethod(method) ||
	1460 dvmIsDeclaredSynchronizedMethod(method))
	1461 {

The ’205 Patent	Infringed By
	1462 LOGW("DexOpt: WARNING: inline op on synchronized method " 1463 "%s.%s\n", 1464 clazz->descriptor, method->name); 1465 /* fail? */ 1466 } 1467 1468 table[tableIndex].method = method; 1469 table[tableIndex].inlineIdx = i; 1470 tableIndex++; 1471 1472 LOGV("DexOpt: will inline %d: %s.%s %s\n", i, 1473 ops[i].classDescriptor, ops[i].methodName, 1474 ops[i].methodSignature); 1475 } 1476 } 1477 } 1478 1479 /* mark end of table */ 1480 table[tableIndex].method = NULL; 1481 LOGV("DexOpt: inline table has %d entries\n", tableIndex); 1482 1483 return table; 1484 } *See also, e.g.*, dalvik\vm\analysis\DexPrepare.c and dalvik\vm\analysis\Optimize.c (*e.g.*, dvmCreateInlineSubsTable routines). The inlineSubs table is a map from Method*’s to indexes into the table returned by `dvmGetInlineOpsTable()`, which is: http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/InlineNative.c#l706 706 /* 707 * Get a pointer to the inlineops table. 708 */ 709 const InlineOperation* dvmGetInlineOpsTable(void) 710 { 711 return gDvmInlineOpsTable; 712 }

The ’205 Patent	Infringed By
	where `gDvmInlineOpsTable` is: http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/InlineNative.c#l628

```
628 /*
629 * Table of methods.
630 *
631 * The DEX optimizer uses the class/method/signature string fields to
decide
632 * which calls it can trample. The interpreter just uses the function
633 * pointer field.
634 *
635 * IMPORTANT: you must update DALVIK_VM_BUILD in DalvikVersion.h if you
make
636 * changes to this table.
637 *
638 * NOTE: If present, the JIT will also need to know about changes
639 * to this table. Update the NativeInlineOps enum in InlineNative.h and
640 * the dispatch code in compiler/codegen/<target>/Codegen.c.
641 */
642 const InlineOperation gDvmInlineOpsTable[] = {
643 { org_apache_harmony_dalvik_NativeTestTarget_emptyInlineMethod,
644 "Lorg/apache/harmony/dalvik/NativeTestTarget;",
645 "emptyInlineMethod", "()V" },
646
647 { javaLangString_charAt,
648 "Ljava/lang/String;", "charAt", "(I)C" },
649 { javaLangString_compareTo,
650 "Ljava/lang/String;", "compareTo", "(Ljava/lang/String;)I" },
651 { javaLangString_equals,
652 "Ljava/lang/String;", "equals", "(Ljava/lang/Object;)Z" },
653 { javaLangString_indexOf_I,
654 "Ljava/lang/String;", "indexOf", "(I)I" },
655 { javaLangString_indexOf_II,
656 "Ljava/lang/String;", "indexOf", "(II)I" },
657 { javaLangString_length,
658 "Ljava/lang/String;", "length", "()I" },
659
660 { javaLangMath_abs_int,
```

The ’205 Patent	Infringed By
	661 "Ljava/lang/Math;", "abs", "(I)I" }, 662 { javaLangMath_abs_long, 663 "Ljava/lang/Math;", "abs", "(J)J" }, 664 { javaLangMath_abs_float, 665 "Ljava/lang/Math;", "abs", "(F)F" }, 666 { javaLangMath_abs_double, 667 "Ljava/lang/Math;", "abs", "(D)D" }, 668 { javaLangMath_min_int, 669 "Ljava/lang/Math;", "min", "(II)I" }, 670 { javaLangMath_max_int, 671 "Ljava/lang/Math;", "max", "(II)I" }, 672 { javaLangMath_sqrt, 673 "Ljava/lang/Math;", "sqrt", "(D)D" }, 674 { javaLangMath_cos, 675 "Ljava/lang/Math;", "cos", "(D)D" }, 676 { javaLangMath_sin, 677 "Ljava/lang/Math;", "sin", "(D)D" }, 678 }; where the elements are instances of: http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/InlineNative.h#l26 26 /* 27 * Basic 4-argument inline operation handler. 28 */ 29 typedef bool (*InlineOp4Func)(u4 arg0, u4 arg1, u4 arg2, u4 arg3, 30 JValue* pResult); ... 44 typedef struct InlineOperation { 45 InlineOp4Func func; /* MUST be first entry */ 46 const char* classDescriptor; 47 const char* methodName; 48 const char* methodSignature; 49 } InlineOperation; The pointers to the functions are references to native instructions, for example, http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/InlineNative.c#l374

The '205 Patent	Infringed By
	374 /* 375 * public int length() 376 */ 377 static bool javaLangString_length(u4 arg0, u4 arg1, u4 arg2, u4 arg3, 378 JValue* pResult) 379 { 380 //LOGI("String.length this=0x%08x pResult=%p\n", arg0, pResult); 381 382 /* null reference check on "this" */ 383 if (!dvmValidateObject((Object*) arg0)) 384 return false; 385 386 pResult->i = dvmGetFieldInt((Object*) arg0, STRING_FIELDOFF_COUNT); 387 return true; 388 } which is compiled into native instructions. The alternative would be to interpret many virtual machine instructions to do the same thing. According to Google's documentation, Dalvik Optimization and Verification With *dexopt* (dalvik\docs\dexopt.html), when an application is downloaded to an Android device, the system installer runs dexopt to create an "optimized DEX" file in the dalvik-cache directory. On an Android system, the PackageManagerService is responsible for installing applications. The Java source code defining the PackageManagerService class is found in PackageManagerService.java. The performDexOptLI() method of the PackageManagerService class is invoked as part of the application installation process. This method in turn invokes the dexopt() method of the Installer class. The PackageManagerService class invokes Installer's dexopt() under other circumstances as well. The Java source code defining the Installer class is found in Installer.java. The dexopt() method of the Installer class, through invocations of execute(), transaction(), writeCommand(),readReply(), and other methods, causes dexopt to run and then returns dexopt's success code to its caller. Because PackageManagerService.java and Installer.java are written in the Java programming language, they are compiled to virtual machine instructions for execution, not native instructions. The dexopt process described in the code above and in the Google documentation

The ’205 Patent	Infringed By
	is performed during the execution of the virtual machine instructions comprising the Installer and PackageManagerService classes. Thus Android satisfies the “at runtime” limitation as construed by the Court.

executing said new virtual machine instruction instead of said first virtual machine instruction.	Android executes the new virtual machine instruction instead of said first virtual machine instruction.
	
	See, e.g.,
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-armv4t.S#l7686
	`7686 /* ------------------------------ */`
	`7687 .balign 64`
	`7688 .L_OP_EXECUTE_INLINE_RANGE: /* 0xef */`
	`7689 /* File: armv5te/OP_EXECUTE_INLINE_RANGE.S */`
	`7690 /*`
	`7691 * Execute a "native inline" instruction, using "/range" semantics.`
	`7692 * Same idea as execute-inline, but we get the args differently.`
	`7693 *`
	`7694 * We need to call an InlineOp4Func:`
	`7695 * bool (func)(u4 arg0, u4 arg1, u4 arg2, u4 arg3, JValue* pResult)`
	`7696 *`
	`7697 * The first four args are in r0-r3, pointer to return value storage`
	`7698 * is on the stack. The function's return value is a flag that tells`
	`7699 * us if an exception was thrown.`
	`7700 */`
	`7701 /* [opt] execute-inline/range {vCCCC..v(CCCC+AA-1)}, inline@BBBB */`
	`7702 FETCH(r10, 1) @ r10<- BBBB`
	`7703 add r1, rGLUE, #offGlue_retval @ r1<- &glue->retval`
	`7704 EXPORT_PC() @ can throw`
	`7705 sub sp, sp, #8 @ make room for arg, +64 bit align`
	`7706 mov r0, rINST, lsr #8 @ r0<- AA`
	`7707 str r1, [sp] @ push &glue->retval`
	`7708 bl .LOP_EXECUTE_INLINE_RANGE_continue @ make call; will return after`
	`7709 add sp, sp, #8 @ pop stack`
	`7710 cmp r0, #0 @ test boolean result of inline`
	`7711 beq common_exceptionThrown @ returned false, handle exception`
	`7712 FETCH_ADVANCE_INST(3) @ advance rPC, load rINST`

The ’205 Patent	Infringed By
	7713 GET_INST_OPCODE(ip) @ extract opcode from rINST 7714 GOTO_OPCODE(ip) @ jump to next instruction This is the computed-goto threaded-interpreter code for the OP_EXECUTE_INLINE_RANGE virtual machine instruction (0xef), which has replaced the OP_INVOKE_DIRECT_RANGE or OP_INVOKE_STATIC_RANGE as the virtual machine instruction. 7708 calls .LOP_EXECUTE_INLINE_RANGE_continue to perform the actual transfer to the native instructions. http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/mterp/out/InterpAsm-armv4t.S#l9502 9502 /* continuation for OP_EXECUTE_INLINE_RANGE */ 9503 9504 /* 9505 * Extract args, call function. 9506 * r0 = #of args (0-4) 9507 * r10 = call index 9508 * lr = return addr, above [DO NOT bl out of here w/o preserving LR] 9509 */ 9510 .LOP_EXECUTE_INLINE_RANGE_continue: 9511 rsb r0, r0, #4 @ r0<- 4-r0 9512 FETCH(r9, 2) @ r9<- CCCC 9513 add pc, pc, r0, lsl #3 @ computed goto, 2 instrs each 9514 bl common_abort @ (skipped due to ARM prefetch) 9515 4: add ip, r9, #3 @ base+3 9516 GET_VREG(r3, ip) @ r3<- vBase[3] 9517 3: add ip, r9, #2 @ base+2 9518 GET_VREG(r2, ip) @ r2<- vBase[2] 9519 2: add ip, r9, #1 @ base+1 9520 GET_VREG(r1, ip) @ r1<- vBase[1] 9521 1: add ip, r9, #0 @ (nop) 9522 GET_VREG(r0, ip) @ r0<- vBase[0] 9523 0: 9524 ldr r9, .LOP_EXECUTE_INLINE_RANGE_table @ table of InlineOperation 9525 LDR_PC "[r9, r10, lsl #4]" @ sizeof=16, "func" is first entry 9526 @ (not reached)

The ’205 Patent	Infringed By
	which at 9524-2525 uses the reference to the table of native instructions to fetch a new native program counter, and transfers to those native instructions.

The '205 Patent	Infringed By

2. The method of claim 1, further comprising overwriting a selected virtual machine instruction with a new virtual machine instruction, the new virtual machine instruction specifying execution of the at least one native machine instruction.	*See* Claim 1, *supra*.
	
	The overwriting of the selected virtual machine instruction with the new virtual machine instruction is in rewriteExecuteInlineRange, cited above, but repeated here for clarity:
	
	http://android.git.kernel.org/?p=platform/dalvik.git;a=blob;f=vm/analysis/DexOptimize.c#l2345
	`2345 /*`
	`2346 * See if the method being called can be rewritten as an inline operation.`
	`2347 * Works for invoke-virtual/range, invoke-direct/range, and invoke-static/range.`
	`2348 *`
	`2349 * Returns "true" if we replace it.`
	`2350 */`
	`2351 static bool rewriteExecuteInlineRange(Method* method, u2* insns,`
	`2352 MethodType methodType, const InlineSub* inlineSubs)`
	`...`
	`2369 insns[0] = (insns[0] & 0xff00) | (u2) OP_EXECUTE_INLINE_RANGE;`
	`2370 insns[1] = (u2) inlineSubs->inlineIdx;`
	
	See also, e.g., dalvik\vm\analysis\Optimize.c.

~~The '205 Patent~~	~~Infringed By~~
~~3. The method of claim 2, wherein the [new virtual machine] instruction includes a pointer to the at least one native machine instruction.~~	~~*See* Claim 2, *supra*.~~ ~~The OP_EXECUTE_INLINE_RANGE bytecode takes as an argument the index of the method in the table of inline subroutines. The first field of each element in that table is a pointer to the native code for that subroutine.~~ ~~`2370 insns[1] = (u2) inlineSubs->inlineIdx;`~~