SpanDex: Secure Password Tracking for Android

Landon P. Cox, *Duke University*
Peter Gilbert, *Duke University*
Geoffrey Lawler, *Duke University*
Valentin Pistol, *Duke University*
Ali Razeen, *Duke University*
Bi Wu, *Duke University*
Sai Cheemalapati, *Duke University*

Abstract

This paper presents SpanDex, a set of extensions to Android's Dalvik virtual machine that ensures apps do not leak users' passwords. The primary technical challenge addressed by SpanDex is precise, sound, and efficient handling of implicit information flows (e.g., information transferred by a program's control flow). SpanDex handles implicit flows by borrowing techniques from symbolic execution to precisely quantify the amount of information a process' control flow reveals about a secret. To apply these techniques at runtime without sacrificing performance, SpanDex runs untrusted code in a data-flow sensitive sandbox, which limits the mix of operations that an app can perform on sensitive data. Experiments with a SpanDex prototype using 50 popular Android apps and an analysis of a large list of leaked passwords predicts that for 90% of users, an attacker would need over 80 login attempts to guess their password. Today the same attacker would need only one attempt for all users.

1 Introduction

Today's consumer mobile platforms such as Android and iOS manage large ecosystems of untrusted third-party applications called "apps." Apps are often integrated with remote services such as Facebook and Twitter, and it is common for an app to request one or more passwords upon installation. Given the critical and ubiquitous role that passwords play in linking mobile apps to cloud-based platforms, it is paramount that mobile operating systems prevent apps from leaking users' passwords. Unfortunately, users have no insight into how their passwords are used, even as credential-stealing mobile apps grow in number and sophistication [12, 13, 24].

Taint tracking is an obvious starting point for securing passwords [11]. Under taint tracking, a monitor maintains a *label* for each *storage object*. As a process executes, the monitor dynamically updates objects' labels to indicate which parts of the system state hold secret information. Taint tracking has been extensively studied for many decades and has practical appeal because it can be transparently implemented below existing interfaces [11, 19, 5, 14].

Most taint-tracking monitors handle only *explicit flows*, which directly transfer secret information from an operation's source operands to its destination operands. However, programs also contain *implicit flows*, which transfer secret information to objects via a program's control flow. Implicit flows are a long-standing problem [8] that, if left untracked, can dangerously understate which objects contain secret information. On the other hand, existing techniques for securely tracking implicit flows are prone to significantly *overstating* which objects contain secret information.

Consider secret-holding integer variable s and pseudocode **if** s != 0 **then** $x := a$ **else** $y := b$ **done**. This code contains explicit flows from a to x and from b to y as well as implicit flows from s to x and s to y. A secure monitor must account for the information that flows from s to x and s to y, regardless of which branch the program takes: y's value will depend on s even when s is non-zero, and x's value will depend on s even when s is zero.

Existing approaches to tracking implicit flows apply static analysis to all untaken execution paths within the scope of a tainted conditional branch. The goal of this analysis is to identify all objects whose values are influenced by the condition. Strong security requires such analysis to be applied conservatively, which can lead to prohibitively high false-positive rates due to variable aliasing and context sensitivity [10, 14].

In this paper, we describe a set of extensions to Android's Dalvik virtual machine (VM) called SpanDex that provides strong security guarantees for third-party apps' handling of passwords. The key to our approach is focusing on the common access patterns and semantics of the data type we are trying to protect (i.e., passwords).

SpanDex handles implicit flows by borrowing tech-

niques from symbolic execution to precisely quantify the amount of information a process' control flow reveals about a secret. Underlying this approach is the observation that as long as implicit flows transfer a safe amount of information about a secret, the monitor need not worry about where this information is stored. For example, mobile apps commonly branch on a user's password to check that it contains a valid mix of characters. As long as the implicit flows caused by these operations reveal only that the password is well formatted, the monitor does not need to update any object labels to indicate which variables' values depend on this information.

To quantify implicit flows at runtime without sacrificing performance, SpanDex executes untrusted code in a data-flow defined sandbox. The key property of the sandbox is that it uses data-flow information to restrict how untrusted code operates on secret data. In particular, SpanDex is the first system to use constraint-satisfaction problems (CSPs) at runtime to naturally prevent programs from certain classes of behavior. For example, SpanDex does not allow untrusted code to encrypt secret data using its own cryptographic implementations. Instead, SpanDex's sandbox forces apps that require cryptography to call into a trusted library.

SpanDex does not "solve" the general problem of implicit flows. If the amount of secret information revealed through a process' control flow exceeds a safe threshold, then a monitor must either fall back on conservative static analysis for updating individual labels or simply assume that all subsequent process outputs reveal confidential information. However, we believe that the techniques underlying SpanDex may be applicable to important data types besides passwords, including credit card numbers and social security numbers. Experiments with a prototype implementation demonstrate that SpanDex is a practical approach to securing passwords. Our experiments show that SpanDex generates far fewer false alarms than the current state of the art, protects user passwords from a strong attacker, and is efficient.

This paper makes the following contributions:

- SpanDex is the first runtime to securely track password data on unmodified apps at runtime without overtainting or poor performance.
- SpanDex is the first runtime to use online CSP-solving to force untrusted code to invoke trusted libraries when performing certain classes of computation on secret data.
- Experiments with a SpanDex prototype show that it imposes negligible performance overhead, and a study of 50 popular, non-malicious unmodified Android apps found that all but eight executed normally.

The rest of this paper is organized as follows: Section 2 describes background information and our motivation, Section 3 provides an overview of SpanDex's design, Section 4 describes SpanDex's design in detail, Section 5 describes our SpanDex prototype, Section 6 describes our evaluation, and Section 7 provides our conclusions.

2 Background and motivation

Under dynamic information-flow tracking (i.e., taint tracking), a monitor maintains a *label* for each *storage object* capable of holding secret information. A label indicates what kind of secret information its associated object contains. Labels are typically represented as an array of one-bit *tags*. Each tag is associated with a different source of secret data. A tag is set if its object's value depends on data from the tag's associated source. *Operations* change objects' state by transferring information from one set of objects to another. Monitors *propagate tags* by interposing on operations that could transfer secret information, and then updating objects' labels to reflect any data dependencies caused by an operation. We say that information derived from a secret is *safe* if it reveals so little about the original secret that releasing the information poses no threat. However, if information is *unsafe*, then it should only be released to a trusted entity.

2.1 Related work: soundness, precision, and efficiency

The three most important considerations for taint tracking are soundness, precision, and efficiency. Tracking is *sound* if it can identify all process outputs that contain an unsafe amount of secret information. Soundness is necessary for security guarantees, such as preventing unauthorized accesses of secret information. Tracking is *precise* if it can identify how much secret information a process output contains. Precision can be tuned along two dimensions: better *storage precision* associates labels with finer-grained objects, and better *tag precision* associates finer-grained data sources with each tag.

Imprecise tracking leads to *overtainting*, in which safe outputs are treated as if they are unsafe. A common way to compensate for imprecise tracking is to require users or developers to *declassify* tainted outputs by explicitly clearing objects' tags.

Tracking is *efficient* if propagating tags slows operations by a reasonable amount. The relationship between efficiency and precision is straightforward: increasing storage precision causes a monitor to propagate tags more frequently because it must interpose on lower-level operations; increasing tag precision causes a monitor to do more work each time it propagates tags. Finding a suitable balance of soundness, precision, and efficiency

is challenging, and prior work has investigated a variety of points in the design space.

One approach to information-flow tracking is to use static analysis in combination with a secrecy-aware type system and programmer-defined declassifiers to prevent illegal flows [20]. This approach is sound, precise, and efficient but is not compatible with legacy apps. Integrating secrecy annotations and declassifiers into apps and platform libraries requires a non-trivial re-engineering effort by developers and platform maintainers.

An alternative way to ensure soundness is to propagate tags on high-level operations that generate only *explicit flows*. An explicit flow occurs when an operation directly transfers information from from a set of well-defined source objects to a set of well-defined destination objects [8]. For example, process-level monitors such as Asbestos [9], Flume [15], and HiStar [23] maintain labels for each address space and kernel-managed communication channel (e.g., file or socket), and propagate tags for each authorized invocation of the system API.

Such process-grained tracking is sound and efficient, but operations defined by a system API commonly manipulate fine-grained objects, such as byte ranges of memory. The mismatch between the granularity of labeled objects and operation arguments leads to imprecision. For example, once a process-grained monitor sets a tag for an address space's label, it conservatively assumes that any subsequent operation that copies data out of the address space is unsafe, even if the operation discloses no secret information.

As with language-based flow monitors, process-grained monitors must rely on trusted declassifiers to compensate for this imprecision. These declassifiers proxy all inter-object information transfers and are authorized to clear tags from labels under their control. However, because declassifiers make decisions with limited context, they can be difficult to write and require developers to modify existing apps.

Other monitoring schemes have improved precision by associating labels with finer-grained objects such as individual bytes of memory [5, 19]. While tracking at too fine a granularity leads to prohibitively poor performance [5, 19] (e.g., 10x to 30x slowdown), propagating tags for individual variables within a high-level language runtime is efficient [11]. The primary challenge for such fine-grained tracking is balancing soundness and precision in the presence of *implicit flows*.

As before, consider secret-holding variable s and pseudo-code **if** s != 0 **then** $x := a$ **else** $y := b$ **done**. Borrowing terminology from [18], we say that all operations between **then** and **done** represent the *enclosed region* of the conditional branch. Thus, the enclosed region contains explicit flows from a to x and from b to y. Operations like conditional branches induce implicit flows by transferring information from the objects used to evaluate a condition to any object whose value is influenced by an execution path through the enclosed region. We refer to the set of influenced objects as the *enclosed set*. The enclosed set includes all objects that are modified along the taken execution path as well as all objects that *might have been modified* along any untaken paths. To ensure soundness, a monitor must propagate s's tags to all objects in the enclosed set.

Propagating tags to members of the enclosed set can lead to overtainting in two ways. First, because a conditional branch does not specify its enclosed set, the membership must be computed through a combination of static and dynamic analysis [5, 18]. In our example, a simple static analysis of the program's control-flow graph could identify the complete enclosed set consisting of x and y. However, strong soundness guarantees require an overly conservative analysis of far more complex untaken paths containing context-sensitive operations and aliased variables. This can overstate which objects' values are actually influenced by a branch. Less conservative tag propagation creates opportunities for malicious code to leak secret information.

Second and more important, the amount of information transferred through a process' control flow is often very low. These information-poor flows expose the problem with tag imprecision. In particular, conventional monitors can only account for an implicit flow by propagating single-bit tags from the branch condition to members of the enclosed set. And yet members of the enclosed set can only reflect as much new information as the branch condition reveals. When the condition reveals very little information (e.g., s != 0), a single-bit tag cannot be used to differentiate between an object whose value is weakly dependent on secret information and one whose value encodes the entire secret. Thus, when an execution's control flow transfers very little information, propagating tags to members of the enclosed set significantly overstates how much secret information the branch transfers to the rest of the program state.

Prior work on DTA++[14] and Flowcheck [18] have articulated similar insights about the causes of overtainting. DTA++ propagates tags to an enclosed set only if an execution's control flow reveals the entire secret (i.e., the execution path is injective with respect to a secret input). However, DTA++ relies on offline symbolic execution of several representative inputs to select which branches should propagate tags to their enclosed sets. Offline symbolic execution provides limited code coverage for moderately complex programs and is unlikely to deter actively malicious programs.

Flowcheck focuses on the imprecision of single-bit taint tags and precisely quantifies the total amount of secret information an execution reveals (as measured in

bits). However, Flowcheck imposes significant performance penalties and must compute the enclosed set (often with assistance from the programmer) to quantify the channel capacity of enclosed regions.

To summarize, we are unaware of any prior work on information-flow tracking that provides a combination of soundness, precision, and efficiency that would be suitable for tracking passwords on today's mobile platforms.

2.2 Android-app study

To test our hypothesis that conventional handling of implicit flows leads to overtainting and false alarms, we created a modified version of TaintDroid [11] called TaintDroid++ that supports limited implicit-flow tracking. TaintDroid and TaintDroid++ track explicit flows the same way. Each variable in a Dalvik executable is assigned a label consisting of multiple tags, and tags are propagated according to a standard tag-propagation logic.

The primary difference between the two monitors is that TaintDroid ignores implicit flows and TaintDroid++ does not. First, for a Dalvik executable, TaintDroid++ constructs a control-flow graph and identifies the immediate post-dominator (ipd) for each control-flow operation. It then uses smali [1] to insert custom Dalvik instructions that annotate (1) each ipd with a unique identifier, and (2) each control-flow operation with the identifier of its ipd. Like Dytan [5], TaintDroid++ does not propagate tags to objects that might have been updated along untaken execution paths.

Using these two execution environments, we ran four popular Android apps that require a user to enter a password: the official apps for LinkedIn, Twitter, Tumblr, and Instagram. Both systems tagged password data as it was input but before it was returned to an app. We then manually exercised each app's functionality and monitored its network and file outputs for tainted data.

Figure 1 shows the number and type of tainted outputs we observed for apps running under TaintDroid and TaintDroid++. For each tainted output, we manually inspected the content to determine whether it contained password data or not. Each tainted output under TaintDroid appeared to be an authentication message that clearly contained a password. TaintDroid++ also tainted these outputs, but generated many more tainted network and file writes. We were unable to detect any password information in these extra tainted outputs, and regard them as evidence of overtainting.

Overtainting is only a problem if incorrectly tainted data is copied to an inappropriate sink. Thus, a false positive occurs when an app copies data that is safe but tainted to an inappropriate sink. Apps authenticate using the OAuth protocol and should not store a local copy

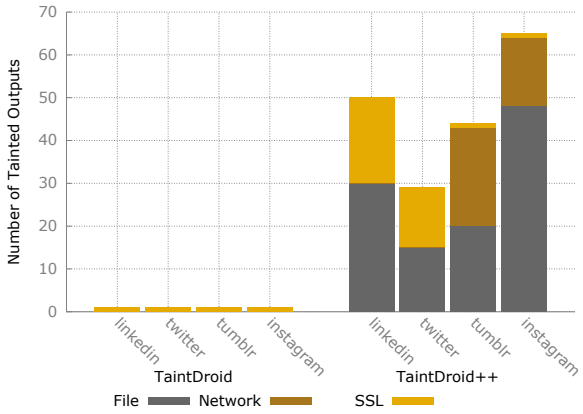

Figure 1: Tainted outputs for apps running under TaintDroid and TaintDroid++.

of a password once they receive an OAuth token from a server. Thus, each tainted file write generated under TaintDroid++ is a false positive.

For network writes, we also consider whether the password data was sent over an encrypted connection (i.e., over SSL) and the IP address of the remote server. Both Tumblr and Instagram under TaintDroid++ generated unencrypted tainted network writes. None of these writes were tainted under TaintDroid. Furthermore, TaintDroid only taints outputs to appropriate servers, but under TaintDroid++ several overtainted outputs were sent to third-parties such as the cloudfront.net CDN and flurry.com analytics servers. These results are consistent with previous work on overtainting [4, 22], and confirm that securing users' passwords requires a better balance of soundness and precision.

3 System Overview

This section provides an overview of SpanDex, including the principles and attacker model that inform its design.

3.1 Principles

SpanDex's primary goal is to soundly and precisely track how information about a password circulates through a mobile app. For example, if an app requests a Facebook password, then SpanDex should raise an alert only if the app tries to send an unsafe amount of information about the password to a non-Facebook server. Preventing leaks also requires a way for users to securely enter and categorize their passwords, and to address these issues we rely on secure password-entry systems such as ScreenPass [17]. SpanDex is focused on tracking information *after* a password has been securely input and handed over to an untrusted app. The following design principles guided our work.

Monitor explicit and implicit flows differently. In practice, explicit and implicit flows affect a program's state in very different ways. Operations on secret data that trigger explicit flows transfer a relatively large amount of secret information to a small number of objects. The inverse is true of control-flow operations that depend on secret data. These operations often transfer very little secret information to members of a large enclosed set. These observations led us to apply different mechanisms to tracking explicit and implicit flows.

First, SpanDex uses conventional taint tracking to monitor explicit flows. SpanDex is integrated with TaintDroid and Android's Dalvik VM, and maintains a label for each program variable. Each label logically consists of a single-bit tag indicating whether the variable contains an unsafe amount of information about a character within a user's password. Because explicit flows transfer a relatively large amount of information between objects, when an object's tag is set, SpanDex assumes that the variable contains an unsafe amount of secret information.

Second, when SpanDex encounters a branch with a tainted condition, it does not immediately propagate tags to objects in the enclosed set. Rather, SpanDex first updates an upper bound on the total amount of secret information the execution's control flow has revealed to that point. This upper bound precisely captures the maximum amount of secret information that an attacker could encode in untagged objects. As long as the total amount of secret information transferred through implicit flows is safe, SpanDex can ignore where that information is stored.

Like DTA++, SpanDex borrows techniques from symbolic execution to quantify the amount of information revealed through implicit flows. In particular, SpanDex integrates operation logging with tag propagation to record the chain of operations leading from a tainted variable's current state back to the original secret input. When SpanDex encounters a tainted conditional branch, it updates its information bounds by using these records to solve a constraint-satisfaction problem (CSP). The CSP solution identifies a set of secret inputs that could have led to the observed execution path. This set precisely captures the amount of information transferred through implicit flows.

The drawback of applying these techniques at runtime is the potential for poor performance. A monitor can efficiently record operations on tainted data at runtime, but solving a CSP when encountering a tainted branch could be disastrous. In the worst case, trying to solve a CSP could cause a non-malicious app to halt. For example, passwords must be encrypted before they are sent over the network, but it is infeasible to compute the set of all plaintext inputs that could have generated an encrypted output. Balancing the need to track implicit flows while preventing common primitives such as cryptography from slowing, or even halting, non-malicious apps led to our second design principle.

If commonly used functionality makes tracking difficult, force apps to use a trusted implementation. Mobile apps typically receive a password, perform sanity checks on the characters, encode the password as an http-request string, encrypt the http-request, and forward the encrypted string to a server. The code used to transform password data from one representation to another (e.g., encoding a character array as an http-request string and then encrypting the string) is problematic because it uses a number of operations that make quantifying implicit flows prohibitively slow or even impossible. This code includes a large number of bit-wise and array-indexing operations interleaved with tainted conditional branches. If SpanDex tracked implicit flows within this code as we have described thus far, non-malicious apps would become unusable.

Fortunately, it is exceedingly rare for apps to implement this functionality themselves. Instead, apps rely on platform libraries for common transformations, such as character encoding and cryptography. On Android this library code is small in size, easy to understand, and protected by the Java type system.

Tracking explicit flows remains the same for trusted libraries as for untrusted app code. However, within a trusted library, SpanDex does not solve CSPs when encountering a tainted branch and may directly update the information bound of a secret before exiting. This approach is sound for library code whose state is strongly encapsulated and whose semantics are well understood.

For example, encrypting a tainted string involves a sequence of calls into a crypto library for initializing the algorithm's state, updating that state, and retrieving the final encrypted result. Ignoring tainted conditional branches within this code is sound for two reasons. First, tracking explicit flows within the library ensures that any intermediate outputs as well as the final output are properly tagged. Second, external code can only access library state through the narrow interface defined by the library API; there is no way for untrusted code to infer properties about the plaintext except those that the library explicitly exposes through its interface or by branching on the plaintext data itself. SpanDex tracks both cases.

The protection boundary separating untrusted code from trusted library code has two novel properties. First, the boundary is defined by both data flow and control flow. An app is allowed to use a custom cryptographic implementation on untainted data, but must use the trusted crypto library to encrypt tainted data. Second, the boundary is enforced by the aggregate complexity of

the operations performed rather than by hardware or a conventional software guard. If an app attempts to encrypt password data using a custom implementation *or* branches on encrypted data returned by the trusted library, it will be forced to solve an intractable CSP and halt.

Thus, the key property of a SpanDex's sandbox is that it restricts the classes of computation that untrusted code may directly perform on secret data. Instead, an app must yield control to the trusted platform so that these computations can be performed on its behalf.

Given an execution environment that can efficiently quantify the amount of secret information transferred through implicit flows, SpanDex's final challenge is determining whether the quantified amount is safe to release. This challenge led to our final design principle.

Use properties of a secret's data type to set release policies. Like SpanDex, DTA++ requires a threshold on the amount of information revealed through implicit flows. DTA++ applies a strict policy to determine when to propagate tags by doing so only when the control flow is injective. That is, DTA++ propagates tags when a single secret value could have led to a particular execution path.

Though simple, this policy is inappropriate for SpanDex. Revealing an entire secret value via implicit flows is clearly unsafe, but revealing partial information about a password may be too. For example, using carefully crafted branches, malware could cause significant harm by narrowing every character of a password to two possible values. However, as we have seen, treating all implicit flows as unsafe leads to prohibitive overtainting. SpanDex's challenge is to support practical release policies that sit between these two extremes.

SpanDex benefits from its focus on passwords. Passwords have a well-defined representation and fairly well understood attacker model. For example, it is reasonable to assume that an attacker knows that a password consists of a sequence of human-readable characters (i.e., ASCII characters 32 through 126), many of which are likely to be alphanumeric. An attacker gains no new information from observing the control flow of a process if the flow reveals that each character is within the expected range of values. We investigate what apps' control flows reveal in Section 6.

3.2 Trust and attacker model

SpanDex is implemented below the Dalvik VM interface (i.e., the Dex bytecode ISA), and the protections provided by this VM provide the foundation for SpanDex's trust model. Most Android app logic is written in Java and compiled into Dex bytecodes, which run in an isolated Dalvik VM instance. SpanDex cannot protect passwords from an app that executes third-party native code *while there is password data in its address space*. Thus, objects tainted with password data must be cleared before an app is allowed to execute its own native code. In addition, once a process invokes third-party native code, it may not receive password data. SpanDex must rely on the kernel to maintain information about which processes have invoked third-party native code. Finally, apps may not write tainted data to persistent storage or send it to another app via IPC.

SpanDex is focused on securely tracking how password data flows within an app. Attacks on other aspects of password handling are outside the scope of our design. First, we assume that users can securely enter their password before it is given to an app, and that users will tag a password with its associated domain. A secure, unspoofable user interface, such as the one provided by ScreenPass [17], can provide such guarantees. Special purpose hardware, such as Apple's Touch ID fingerprint sensor and secure enclave [2], could also provide this guarantee.

Second, SpanDex can help ensure that password data is shared only with servers within the domain specified by the user, but provides no guarantees once it leaves a device. For example, SpanDex cannot prevent an attacker from sending a user's Facebook password as a message to a Facebook account controlled by the attacker. Preventing such cases requires cooperation between SpanDex and the remote server. SpanDex could notify the service when a message contains password data, and the service could determine whether such messages should contain password data.

We assume that an attacker completely controls one or more apps that a user has installed, and that the attacker is also in control of one or more remote servers. The attacker's servers can communicate with the attacker's apps, but the servers reside in a different domain than the one the user associates with her password. The attacker can make calls into the platform libraries and manipulate its apps' data and control flows to send information about passwords to its remote servers.

Based on the large-scale leakage of large password lists from major services, such as Gawker [21] and Sony Playstation [3], we assume that an attacker has access to a large list of unique passwords, and that the user's password is on the list. However, we assume that the attacker does not know which usernames are associated with each entry in its list (though it does know the user's username).

Thus, our attacker's goal is to de-anonymize the user within its password list using information gathered from its apps. The attacker can send its servers as much untainted data describing a user's password as SpanDex's release policies allow (i.e., the password length as well

as a range of possible values for each password character). In the worst case, the attacker will eliminate all but one of the passwords in its list. On the other hand, if the app provides no new information, then the user's password could be any on the list.

Once the attacker has computed the set of possible passwords for a username, it can only identify the correct username-password combination through online querying. For example, if an attacker infers that Bob's Facebook password is one of ten possibilities, then the attacker needs at most ten tries to login to Facebook as Bob.

The attacker may also have extra information about the usage distribution of passwords in its database. For example, the attacker may know that one password is used by twice as many users as another. While information from the app can help the attacker narrow a user's password to a smaller set of possibilities, the usage distribution allows the attacker to prioritize its login attempts to reduce the expected number of attempts before a successful login. We return to this issue in Section 6.

4 SpanDex

As with conventional taint tracking, SpanDex updates objects' labels on each operation that generates an explicit flow. If the monitor encounters a control-flow operation with a tainted condition, it does not update the labels of objects in the enclosed set. Instead, the monitor updates an upper bound on the amount of information the execution's control flow has revealed about the secret input.

SpanDex represents this bound as a *possibility set (p-set)*. SpanDex maintains a p-set for each password character an app receives. P-sets logically contain the possible values of a character revealed by a process' control flow. Each time the app's control flow changes as a result of tainted objects, SpanDex attempts to remove values from the secret's p-set.

4.1 Operation dag

In order to narrow a p-set, SpanDex must understand the data flow from the original secret values to a tainted condition. We capture these dependencies in an *operation dag (op-dag)*. This directed acyclic graph provides a record of all taint-propagating operations that influenced a tainted object's value as well as the order in which the operations occurred.

SpanDex reuses TaintDroid's label-storage strategy, and stores each 32-bit label adjacent to its object's value. However, whereas each bit in a TaintDroid label represents a different category of sensitive data (e.g., location or IMEI), SpanDex labels are pointers to nodes in the op-dag. If an object's label is null, then it is untainted. If an object's label is non-null, then its value depends on secret data.

Label storage in SpanDex most significantly differs from TaintDroid for arrays. In TaintDroid, each array is assigned a single label for all entries. If any array element becomes tainted, then the entire array is treated as tainted. This approach is inappropriate for SpanDex because we want to track individual password characters. Thus, SpanDex maintains per-entry labels. However, the reason that TaintDroid maintains a single label for each array is storage overhead. Byte and character arrays account for a large percentage of an app's memory usage, and assigning a 32-bit label for each byte-array entry could lead to a minimum fourfold increase in memory overhead for array labels.

To avoid this overhead, SpanDex allocates labels for arrays only after they contain tainted data. Each array is initially allocated a single label. If the array is untainted, then its label points to null. If the array contains tainted data, then its label points to a separate label array, with one label for each array entry. As with local-variable and object-field labels, array-element labels point to nodes in the op-dag. Since very few arrays contain password data, the overhead of maintaining per-entry labels is low overall.

The roots of the op-dag are special nodes that contain the original value of each secret (i.e., each password character), a pointer to the secret's p-set, and domain information. A p-set is represented as a doubly-linked list of value ranges. Each entry in the list contains a pointer to the previous and next entries, as well as a minimum and maximum value. Ranges are inclusive, and the union of the ranges specifies the set of possible secret values revealed by an app's control flow. SpanDex initializes p-sets to the range $[32, 126]$ to represent all printable ASCII characters. A secret's domain can be specified by the user through a special software keyboard [17].

Each tainted object version has an associated non-root node that records the operation that created the version, including its source operands. Source operands can be stored as concrete values (when operands are untainted) or as pointers to other nodes in the op-dag (when operands are tainted).

A node can point to more than one node, and there may be multiple paths from a node to one or more roots. The more complex the paths from a node to the op-dag roots are, the more complex updating p-sets becomes.

4.2 Example execution

If a tainted variable influences an app's control flow (e.g., via a conditional branch), then SpanDex traverses the op-dag from the node pointed to by the object's label toward

the roots. To demonstrate how SpanDex maintains and uses op-dags and p-sets, consider the simple snippet of pseduo-code below. Figure 2 shows the resulting op-dag and p-set.

```
0000: mov v1, v0          // v0, v1 label=ROOT
0002: add v2, v1, 3       // v2's label=N1
0004: add v2, v2, 2       // v2's label=N2
0006: sub v3, 6, v2       // v3's label=N3
0008: add v2, v2, 7       // v2's label=N4
000a: const/16 v4, 122    // v4's label=0
000c: if-le v3, v4, 0016
000e: ...
```

The first character of the password is 'p', or numeric value 112, and is stored in register v0. The password's domain is Facebook. v0's label points to the Root node for the secret character. v0 is then copied into v1, whose label must also point to Root. The sum of v1 and 3 is then stored in v2, whose label then points to new node, N1. N1 contains the addition operation, the 3 operand, and points to Root. The next line adds 2 to v2. This creates a new version of v2, which is recorded in N2. N2 contains the 2 operand and points to the node for the previous version of v2, node N1. The remaining arithmetic operations proceed similarly. Finally, the code loads the constant value of 122 into v4 for an upcoming conditional branch. v4's label is null, since it is not tainted.

When the code reaches the conditional branch, v3 is less than or equal to v4, since v3's value is 111, and v4's value is 122. Because v3's label is non-null, SpanDex uses the op-dag node in v3's label (N3) to update the p-set.

Updating the p-set is equivalent to solving a CSP to determine which secret values could have led to the control-flow change. In our example, updating the p-set is easy. SpanDex solves the inequality $v0+6-2-3 \leq 122$, leading to $v0 \leq 121$. Thus, the control flow reveals that the first character of the user's password is within the range of $[32, 121]$. SpanDex updates the p-set to reflect this before resuming execution. Figure 2 shows the state of the op-dag and p-set at this point.

This simple example demonstrates some of the challenges and nuances of SpanDex's approach. First, each node in the op-dag represents a version of a tainted variable. N3 points to the version of v2 used to update v3, so that when SpanDex reaches the conditional branch, it can retrieve the sequence of operations that led to v3's current value.

Second, reversible operations such as addition and subtraction make updating p-sets straightforward. Unfortunately, Dalvik supports a number of instructions that are much trickier to handle. For example, Dalvik supports instructions for operating on Java Object references and arrays that behave very differently than simple arithmetic operations. Even some classes of arithmetic operations, such as bit-wise operators and division, can make solving a CSP non-trivial.

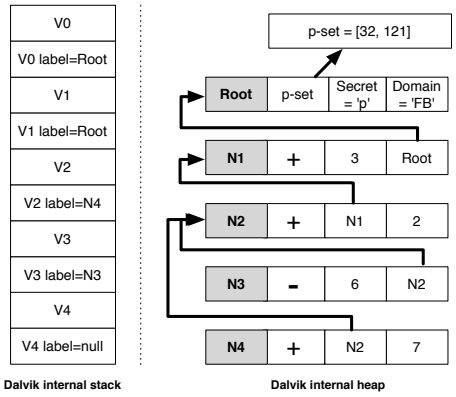

Figure 2: Simple op-dag and p-set example.

Third, there was a single path from N3 to Root in our example. If N3 had forked due to multiple tainted operands, or had led to multiple root nodes due to mixing secret characters, solving the CSP would have been far more complex. Compression and cryptography often mix information from multiple characters, which creates a complex nest of paths from nodes to the op-dag roots.

Fortunately, among the popular non-malicious apps that we have studied, difficult-to-handle operations occur only in platform code such as the Android cryptography library. Furthermore, it is rare to find apps that branch on the results of these operations outside of platform code. Thus, as long as SpanDex can ensure that all outputs from these libraries are explicit and tainted, then we can ignore implicit flows within them (and, thus, avoid CSP solving).

This approach is intuitive. First, outside of simple sanity checking on a password, there is little reason for an app to operate on password data itself. Second, libraries such as a crypto library are designed to suppress implicit flows. Observing an encrypted output or a cryptographic hash should not reveal anything about the plaintext input. Third, there is no obvious reason why app code should branch on either encrypted data or a cryptographic hash. Apps simply use the platform libraries to encode these outputs as strings and send them to a server.

There are many difficult operations that we have not observed in either app code or library code. Our general approach to these operations is to propagate taint to the results of these operations, but to fault if they cause the control flow to change. For example, an app may use bit-wise operations to encode a character, but branching on the encoded result is not allowed. This is secure and does not disrupt non-malicious apps. In the next section, we describe how SpanDex treats each class of Dex bytecodes in greater detail.

4.3 Dex bytecodes

In this section, we describe how SpanDex handles each of the following classes of bytecodes: type-conversion operations, object operations, control-flow operations, arithmetic operations, and array operations.

Type conversions. Dalvik supports the following data types: boolean, byte, char, short, int, long, float, double, and Object reference, as well as arrays of each of these types. P-set ranges are represented internally as pairs of floats. Solving CSPs involving conversions to alternate representations is supported as long as the type is a native and numeric.

Object operations. Dex provides a number of instructions for converting between data types, but conversions can also occur through Object-method invocations and arrays. For example, an app could index into an Object array with a tainted character, where the fields of each Object encodes its position in the array. The returned Object reference would be tainted, and would identify the character used to index into the array. The return value of any method used to access a field of the tainted Object would also be tainted. However, SpanDex would have to understand the internal semantics of the Object in order to solve a CSP involving the tainted returned value. Thus, branching on data derived from a tainted Object reference is not allowed.

Control-flow. A Dalvik program's control flow can change as a result of secret data in many ways. Conditional branch operations such as `if-eq` are the most straightforward, and SpanDex handles these as described in Section 4.2.

Dalvik also supports two case switching operations: `packed-switch` and `sparse-switch`. Both instructions take an index and a reference to a jump table as arguments. The difference between the instructions is the format of the jump table and how it is used. The table for a `packed-switch` is a list of key-target pairs, in which the keys are consecutive integers. Dalvik first checks to see if the index is within the table's range of consecutive keys. If it is not, then the code does not branch and execution resumes at the instruction following the switch instruction. If it is in the table, then the code computes the new PC by adding the matching target to the current PC.

The table for a `sparse-switch` is also a list of key-target pairs, but the keys do not have to be consecutive integers (though they have to be sorted from low-to-high). To handle this instruction, the VM checks whether the index is greater than zero and less than or equal to the table size. It then uses the index to perform a binary search on the keys to find a match. If it finds a match, then it jumps to the instruction at the sum of the matching target and current PC.

Although more complex than conditional branches, handling these switch instructions is straightforward. If the code falls through the switch instruction, then the resulting implicit flow reveals that the index is not equal to any of the table keys. SpanDex can solve a CSP for each of the keys and update its p-sets accordingly. If the control-flow is diverted by the switch instruction, then the resulting implicit flow reveals that the index is equal to the matching table key. SpanDex can solve a CSP for this condition as well. In practice, most switch instructions are packed and the corresponding jump tables are small, which makes solving CSPs for these operations fast.

Finally, a program's control flow can be influenced by tainted data if an operation on tainted data causes an exception to be thrown. For example, an app could divide a number by a tainted variable with a value of zero, or it could use a tainted variable to index beyond the length of an array. SpanDex could compute a CSP for the information revealed by each of these conditions, e.g., that a tainted variable is equal to zero or that a tainted variable is greater than the length of an array. However, we have not seen this behavior in any of the apps we have studied. As a result, our current implementation simply stops the program when an instruction with a tainted operand causes an exception to be thrown.

Arithmetic. As we saw in Section 4.2, reversible arithmetic operations are straightforward to handle. Other arithmetic operations are not impossible to handle, but require a complex solver. For example, reversing multiplication and division operations is tricky because of rounding. Bit-wise operations are even more difficult to reason about. Fortunately, it is exceedingly rare for app code to branch on the results of these operations. Instead, we have observed that trusted library code is far more likely to branch on the results of these operations. As long as we can ensure that all library outputs are explicit, then we do not need to solve CSPs involving difficult operations when in trusted code.

Arrays. Dex provides instructions for inserting (`iput`) and retrieving (`iget`) data from an array. Due to type-conversion problems, SpanDex does not allow tainted indexing of non-numeric arrays. In particular, an app may not use a tainted variable to index into an Object array.

Handling an `iget` operation requires keeping a checkpoint of the array in the op-dag node for the variable holding the result. For example, say that all of the entries in an int array are zero or one, and that an app indexes into the array with a tainted variable. The returned value would be stored in a tainted variable. If the app later branched on the tainted variable, then SpanDex must look at the array checkpoint to determine which indexes would have returned the same value as the exe-

cuted `iget`. In practice, tainted `iget` instructions are rare, and when they do occur the arrays are small.

Unlike a tainted `iget`, a tainted `iput` instruction is dangerous. Consider an attacker that initializes an array a with known size, such that all entries are equal to zero. It then stores the first password character in the variable s and inserts a one into $a[s]$. Because SpanDex maintains per-entry labels for arrays, $a[s]$ is tainted, but no other entries are. The attacker can then incrementally send each value in the array to its server: only $a[s]$ is tainted and will be stopped by SpanDex. Unfortunately, stopping the app at this point is too late, since the number of received zeros reveals the value of s. As a result of this attack, tainted `iput` instructions are illegal.

Finally, Dex also provides instructions such as `filled-new-array` for creating and populating arrays, and SpanDex disallows tainted operands on these instructions.

4.4 Trusted libraries

As described above, there are a number of operations on tainted data that would add significant complexity to SpanDex's CSP solver to support. Even worse, the complexity of the op-dags that combinations of these operations would create make it doubtful that even a sophisticated solver could handle them quickly, if at all. Ideally, these operations would never arise, and if they did, an app would never branch on their results. Sadly, this not the case. Many apps require cryptographic and string-encoding libraries to handle passwords, and these libraries are rife with difficult to handle operations as well as branching on the results of those operations.

Trying to solve such complex CSPs would make SpanDex unusable: non-malicious apps would halt just trying to encrypt a password. At the same time, ignoring flows generated by these operations is not secure. Luckily, we have observed that branching on the results of difficult operations consistently occurs within a handful of simple platform libraries.

Thus, SpanDex's approach to handling difficult implicit flows is to identify the functionality that creates them in advance and to isolate these flows inside trusted implementations. As long as the outgoing information flows from these libraries are always tainted and explicit, SpanDex does not need to worry about their internal control-flow leaking secret information. Furthermore, this code is open and well known, is protected by the Java type system, and can be modified to eliminate implicit flows through the library API.

The set of libraries that SpanDex trusts not to leak information implicitly is: java.lang.String (selected methods excluded), java.lang.Character, java.lang.Math, java.lang.IntegralToString, java.lang.RealToString, java.lang.AbstractStringBuilder, java.net.URLEncoder, java.util.HashMap, android.os.Bundle, android.os.Parcel, and org.bouncycastle.crypto. Nearly all of this code is either stateless string encoding and decoding or cryptography.

4.5 Various attacks and counter-measures

We described several attacks in Section 3.2 that are beyond the scope of SpanDex. In this section, we describe several other attacks and how SpanDex might handle them.

First, SpanDex does not allow tainted data to be written to the file system or copied to another process via IPC. This is reasonable because mobile apps should only require a user's password to retrieve an OAuth token from a remote server. After receiving the token, the app should discard the user's password. If an app tries to copy tainted data to an external server, then SpanDex must consult the domains in the set of reachable op-dag root nodes.

Second, an attacker could have multiple apps under its control generate multiple overlapping (but not identical) p-sets. Each individual p-set would appear safe, but when combined at the attacker's server, they could collectively reveal an unsafe amount of information. Relatedly, a malicious app could request a user's password multiple times and compute different ranges on each password copy.

One way to detect this class of attacks is by inspecting the membership of a secret's p-sets. For the apps that we have observed, p-sets usually correspond to natural character groupings, e.g., numbers, lower-case letters, upper-case letters, and related special characters. P-sets containing unusual character groupings could be a strong signal that an app is malicious.

The solution to this attack suggests a larger class of counter-measures that use information from the p-sets and op-dag to detect malicious behavior. For example, anomalous operation mixes or an unusually large op-dag could indicate an attack. One of the advantages of SpanDex is that it gives the monitor a great deal of insight into how an app operates on password data. We believe that this information could enable a rich universe of policies, though enumerating all of them is beyond the scope of this paper.

Finally, it is possible that SpanDex is vulnerable to certain classes of side-channel and timing attacks that we have not considered. However, any attack that relies on branching on tainted data would be detected. For example, consider the well-known attack on Tenex's password checker [16]. Even though the attack uses a page-fault side channel that is out of SpanDex's scope, SpanDex would have prevented it because each additional charac-

ter comparison would have narrowed its p-set to an unsafe level.

5 Implementation

Our SpanDex prototype is built on top of TaintDroid for Android 2.3.4. We modified TaintDroid to support p-sets and op-dags, and made several modification to the Android support libraries. Most of our changes to these libraries were made in java.lang.String.

First, public String methods whose return value could reveal something about a tainted string's value are not considered trusted to ensure that p-sets are updated properly (e.g., equals(Object), compareTo(String)).

Second, as a performance optimization, the Dalvik VM replaces calls to certain performance-critical Java methods with inlined "intrinsic" functions that are written in C and built in to the Dalvik VM (e.g., String.equals(Object), String.compareTo(String)). However, if an intrinsic inlined function operates on a tainted string and performs comparisons involving the string's characters, we are unable to update the p-sets accordingly. To avoid this, we modified Dalvik's intrinsic inlines that operate on strings to check if the string is tainted and, if so, invoke the Java version instead.

Third, Android's implementation of java.lang.String performs an optimization when converting an ASCII character to its String value: it uses the character's ASCII code to index into a constant char array containing all ASCII characters. If the character to be converted is tainted, we prevent this optimization from being used, as it would result in an array lookup with a tainted index.

Finally, we modified the android.widget.TextView and implemented a custom IME with a special tainted input mode that can be enabled to indicate to SpanDex when a sequence of characters is sensitive (i.e., a password).

6 Evaluation

In order to evaluate SpanDex, we sought answers to the following questions: How well does SpanDex protect users' passwords from an attacker? What is the performance overhead of SpanDex?

6.1 Password protection

As described in Section 3.2, we have designed SpanDex based on an attacker that has access to a large list of cleartext passwords. The attacker knows that a user's password is in the list, and uses untainted information from its malicious app to narrow a user's password to a smaller set of possibilities. To understand how well SpanDex can protect users from such an attack, we need to know the kind of p-sets that real apps induce, we need access to a large list of cleartext passwords, and we need a realistic distribution of how passwords are used. All of these pieces of information will allow us to calculate the number of expected logins an attacker would need to guess a user's password, given the amount of untainted password information that SpanDex allows apps to reveal.

First, we ran 50 popular apps from Google's Play Store. Each of these apps required a login, and we used the same 35-character password for each app. The password contained one lower-case letter ("a"), one upper-case letter ("A"), one number ("0"), and one of each of the 32 non-space special ASCII characters. 42 ran without modification[1]. The top row of Table 1 shows each character in the password.

Eight apps invoked native code before requesting a user's password[2]. While these apps would have to be modified to run under SpanDex, waiting to invoke native code before requesting a user's password is unlikely to require major changes. All other apps ran normally.

For the 42 apps that ran unmodified, after their password was sent, we inspected the p-set for each password character and counted its size. Table 1 shows the maximum, 75th percentile, median, 25th percentile, and minimum p-set size for each password character. The header of the table shows the password. The first thing to notice is that the p-sets for the letters in our password (i.e., "A" and "a") were never smaller than 26. This makes sense, since each app is branching to determine that the character is either a lower or upper case letter. The same is true for the number in our password, "0". No numeric p-set was smaller than 10.

The more difficult cases are the non-alphanumeric special characters. For these cases, the p-sets are fairly app specific. In some cases, the app's control flow depends on a specific character (e.g., Skout with several special characters), but most characters' p-sets remain large across most apps. With the exception of "*", "-", ".", and "_", all non-alphanumeric characters had large p-sets for 75% of apps or more.

Given this observed app behavior, we next obtained the uniqpass-v11 list of 131-million unique passwords [7]. The list contains passwords from a number of sources, including the Sony [3] and Gawker [21] leaks. To simulate an attack, we selected a password, p, from the list and computed the p-sets that a typical app would generate for p. In particular, we assume that the at-

[1] Audible, Amazon, Amazonmp3, Askfm, Atbat, Badoo, Chase, Crackle, Ebay, Etsy, Evernote, Facebook, Flipboard, Flixster, Foursquare, Heek, Howaboutwe, Iheartradio, imdb, LinkedIn, Myfitnesspal, Nflmobile, Pandora, Path, Pinger, Pinterest, Rhapsody, Skout, Snapchat, Soundcloud, Square, Tagged, Textplus, Tumblr, Tunein, Twitter, Walmart, Wordpress, Yelp, Zillow, Zite, and Zoosk

[2] Dropbox, Hulu+, Kindle, Mint, Skype, Spotify, Starbucks, and Voxer

	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/	0	:	;	<	=	>	?	@	A	[	\	]	^	_	‘	a	{		}	~
Max	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95	95
75th	90	16	33	90	90	33	33	90	90	90	90	90	90	90	83	92	90	90	33	90	92	90	90	90	90	32	65	90	90	90	95	90	90	90	90
Med	16	12	12	16	16	12	12	16	16	13	16	16	13	13	14	10	7	7	7	7	7	7	7	26	6	6	6	6	6	6	90	4	4	4	4
25th	12	4	12	12	12	12	12	12	12	1	12	12	1	1	12	10	7	7	7	7	7	7	7	26	5	5	5	5	1	5	26	4	4	4	4
Min	1	1	3	1	1	1	1	1	1	1	3	4	1	1	1	10	1	1	1	1	1	1	3	26	1	1	2	4	1	4	26	1	1	3	4

Table 1: Password-character p-set sizes for 42 popular Android apps

tacker can infer p's length and whether each character is a lower-case letter (26 possibilities), an upper-case letter (26 possibilities), a number (10 possibilities), or a member of a block of special ASCII characters (i.e., the 16 characters below “0”, the seven characters between “9” and “A”, the six characters between “Z” and “a”, and the four characters after “z”).

This information gave us a kind of regular expression for p based on the type of each of its characters. We call the set of passwords matching this expression the *match set* and the size of the match set the *match count*. The larger a password's match count, the more uncertain an attacker is about what password the user entered. We computed the match count for all passwords in the uniqpass list in this way. Finally, we counted the number of passwords with a given match count to arrive at the inverse distribution function.

These calculations show that if SpanDex allows an attacker to learn the p-sets for a password from a typical app, the attacker will have trouble narrowing the set of possible passwords for the user. In particular, 92% of passwords have a match count greater than 10,000, 96% of passwords have a match count greater than 1,000, 98% of passwords have a match count greater than 100, and 99% of passwords have a match count greater than 10.

Unfortunately, recent work on a variety of password databases suggest that password usage follows a zipf distribution [6]. Thus, we also model the N passwords in a match set as a population of N elements that contains exactly one success (as a user would only have one correct password). Next, we let n be the random variable denoting the number of tries required to guess the correct password and find $E[n]$, the expected value of n. If the passwords are all equally probable, we try them in random order. Otherwise, we try them in the descending order of their probability. Note that each password try is done without replacement, i.e., after trying i passwords, we only consider the remaining $(N-i)$ passwords when picking the next most probable password.

A study of the distribution of passwords publicly leaked from Hotmail, flirtlife.de, computerbits.ie, and RockYou found that the passwords in each of these sets can be reasonably modelled by a zipf distribution with s parameter values of 0.246, 0.695, 0.23, and 0.7878 respectively [6]. Using these values of s, we modeled the passwords in each match set and computed the CDF of $E[n]$.

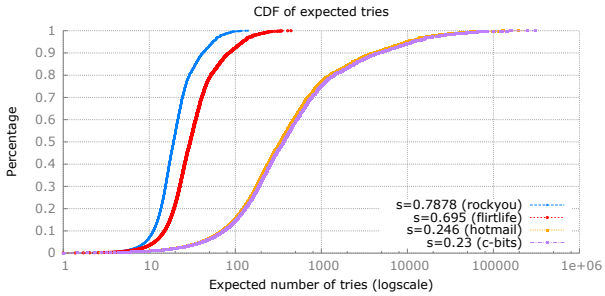

Figure 3: CDFs of expected login attempts using the uniqpass password list.

When $s = 0.7878$, 95% of the time, the attacker is likely to guess the correct password within 50 tries. When the s value for the zipf distribution is 0.246 or less, 99% of passwords are expected to require 10 or more login attempts, and 90% of passwords are expected to require 80 or more attempts. Figure 3 shows the CDFs for all four s values.

Unfortunately, we do not know the usage distribution for the uniqpass dataset since it contains only unique passwords.

6.2 Performance overhead

To measure the performance overhead of SpanDex we used the CaffeineMark benchmark and compared it to stock Android 2.3.4 and TaintDroid. Both TaintDroid and SpanDex ran without any tainted data. Since SpanDex only handles password data that is discarded after an initial login, this is SpanDex's common case. The benchmark was run on a Nexus S smartphone. The results are in Figure 4.

Overall, SpanDex performs only 16% worse than stock Android and 7% worse than TaintDroid. Stock Android performs significantly better than either TaintDroid or SpanDex in the string portion of the benchmark. This is because TaintDroid and SpanDex both disable some optimized string-processing code to store labels.

Finally, we would like to note that when testing apps in Section 6.1, we did not encounter any noticeable slow down under SpanDex. This was due to login being dominated by network latency and the simplicity of the CSPs these apps generated.

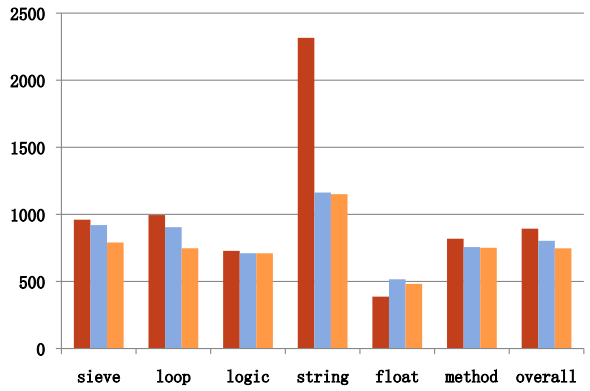

Figure 4: CaffeineMark results for Android (left bar), TaintDroid (middle bar), and SpanDex (right bar).

7 Conclusion

SpanDex tracks implicit flows by quantifying the amount of information transferred through implicit flows when an app executes a tainted control-flow operation. Using a strong attacker model in which a user's password is known to exist in a large password list, we found that for a realistic password-usage distribution, for 90% of users an attacker is expected to need 80 or more login attempts to guess their password.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments. Our work was supported by Intel through the ISTC for Secure Computing at UC-Berkeley as well as the National Science Foundation under NSF awards CNS-0747283 and CNS-0916649.

References

[1] smali: An assembler/disassembler for android's dex format, 2013.

[2] Apple. iphone 5s, 2013.

[3] P. Bright. Sony hacked yet again, plaintext passwords, e-mails, dob posted, 2011.

[4] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA '08*, 2008.

[5] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07*, 2007.

[6] David Malone and Kevin Maher. Investigating the Distribution of Password Choices. In *WWW '12*, April 2012.

[7] Dazzlepod. Uniqpass v11, 2013.

[8] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[9] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP '05*, 2005.

[10] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, 2010.

[12] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys '12*, 2012.

[13] X. Jiang. Smishing vulnerability in multiple android platforms, 2012.

[14] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS '11*, 2011.

[15] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP '07*, 2007.

[16] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17(5):33–48, Oct. 1983.

[17] D. Liu, E. Cuervo, V. Pistol, R. Scudellari, and L. P. Cox. Screenpass: Secure password entry on touchscreen devices. In *MobiSys '13*, 2013.

[18] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6):193–205, June 2008.

[19] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS '05*, 2005.

[20] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[21] L. Segall. Gawker data exposed in major hack attack, 2010.

[22] A. Slowinska and H. Bos. Pointless tainting? evaluating the practicality of pointer tainting. In *EuroSys '09*, 2009.

[23] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI '06*, 2006.

[24] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *IEEE SP '12*, 2012.