Communix: A Framework for Collaborative Deadlock Immunity

Horatiu Jula, Pınar Tözün, George Candea
École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland
Email: {horatiu.jula,pinar.tozun,george.candea}@epfl.ch

***Abstract*—We present Communix, a collaborative deadlock immunity framework for Java programs. Deadlock immunity enables applications to avoid deadlocks that they previously encountered. Dimmunix [1], our deadlock immunity system, detects deadlocks and saves their signatures at runtime, then avoids execution flows that match these signatures; a signature is an abstraction of the execution flow that led to deadlock. Dimmunix needs all the deadlock bugs in an application to manifest, in all possible ways, in order to provide full protection against deadlocks for that application. Communix addresses this shortcoming by distributing the deadlock signatures produced by Dimmunix. The signatures of a deadlock can protect against the deadlock any user connected to the Internet and running the same application, even if he/she did not experience the deadlock yet. Besides signature distribution, Communix provides signature validation and generalization. Signature validation ensures that the incoming signatures match the target applications, and protect the users against malicious signatures. Signature generalization keeps the repository of deadlock signatures compact, by merging multiple deadlock signatures into one signature. Communix is application agnostic, i.e., it is applicable to any Java application. Communix is efficient and scalable, and can effectively protect Java applications against malicious signatures.**

I. Introduction

Failure immunization techniques protect the programs against a specific bug or vulnerability exploit by learning from its past manifestations. We use the term "failure" to denote a manifestation of the bug. An immunization system detects the failure, extracts its fingerprint, and uses it to avoid reoccurrences of the same bug. We call this fingerprint "bug signature". A bug signature is an approximation of the execution flow that led to the failure. For instance, Dimmunix detects deadlocks at runtime and generates signatures to avoid reoccurrences of the same deadlocks.

Failure immunization systems avoid only manifestations of previously encountered bugs; a bug must manifest at least once for the application to be protected against it. Therefore, the false positives rate is low, because only manifestations of real bugs are avoided. A false positive is the situation where a failure is avoided with no reason, i.e., the failure could not have occurred, even without any avoidance. However, there are false negatives (i.e., bugs against which the application is unprotected) until all the bugs manifest.

One possible solution to address the aforementioned drawback is collaborative immunization via distribution of bug signatures. More specifically, once a user encounters a bug, the bug's signature is automatically generated and distributed to other users through the Internet. Therefore, each bug needs to be encountered once, by any user, then the other users get protected against the bug, without having to experience it.

We present Communix, a collaborative immunization framework that enables Java programs running on different machines to immunize each other against deadlock bugs. Communix provides three services: signature distribution, signature validation, and signature generalization.

To distribute deadlock signatures, Communix uses an immunity server; client machines upload signatures discovered by Dimmunix to the server, and periodically retrieve the new signatures from the server. Each time a Java application starts on a client machine, Communix selects from these signatures the ones that are valid for that application and, if possible, it generalizes existing deadlock signatures.

Communix is efficient and scalable. In §IV, we show that the server can process efficiently 30,000 simultaneous requests, at a rate of 9,000 requests per second. The agent can analyze 1,000 new deadlock signatures in 2-3 seconds (§IV).

We present two scenarios that illustrate the benefit of frameworks like Communix. In the first scenario, the user opens a web page, and the browser deadlocks while rendering the content of the page, due to a Java applet. The user shuts down the browser, then restarts it and opens the same page. If the browser is equipped with Dimmunix [1], it will successfully open the page; if not, it might deadlock again. However, it may be undesirable to have the browser deadlocking in the first place. Even the first occurrence of the deadlock may have severe consequences: the browser might be in the middle of some important operation, like purchasing an expensive product, or booking a flight. Therefore, a framework like Communix that prevents other users from encountering the deadlock in the first place is beneficial. In the second scenario, a deadlock-prone version of a plugin is released for the Eclipse IDE, which makes Eclipse hang. If the plugin has multiple deadlock bugs, each user has to encounter all these deadlocks for Dimmunix to be able to avoid them. Sharing the signatures of the deadlocks with users who just installed the plugin is useful; these users will not experience any deadlocks while using the plugin if all deadlocks have already been encountered by some users.

The contributions of this work are: First, Communix transparently distributes signatures over the Internet, to prevent other users from encountering deadlock bugs. Second,

Appears in Proceedings of the Intl. Conf. on Dependable Systems and Networks (DSN), Hong Kong, China, June 2011

Communix finds concise signatures of a deadlock by using the collective knowledge of all the nodes in the Internet that run the same application. Communix provides the above features while protecting the Java applications against DoS attacks that attempt to exploit Dimmunix by providing fake deadlock signatures.

This paper is organized as follows: We provide background information in §II, describe the design of Communix in §III, evaluate Communix in §IV, present related work in §V, and conclude in §VI.

II. Background

In this section, we briefly present Dimmunix (§II-A), and introduce concepts related to collaborative deadlock immunity (§II-B).

A. Dimmunix

Programs augmented with Dimmunix develop antibodies against each deadlock they encounter: Dimmunix extracts the signature of the deadlock, stores it in a persistent history, then alters future thread schedules transparently to the application, in order to avoid execution flows matching the signature. A signature approximates the execution flow leading to deadlock. With every newly encountered deadlock, the program's resistance to deadlocks is improved. Hence, deadlock signatures constitute effective antibodies against deadlock bugs.

A deadlock signature consists of (1) the call stacks the deadlocked threads had when they acquired the locks involved in the deadlock and (2) the call stacks of the deadlocked threads at the moment of the deadlock. We call the former "outer call stacks" and the latter "inner call stacks"; we call the top frames of these call stacks "outer" and respectively "inner" lock statements. A deadlock bug is uniquely delimited by the outer and inner lock statements. Dimmunix used to keep only the outer call stacks in a signature [1]; we made it keep also the inner call stacks, in order to have an accurate localization of the deadlock bug within the signature. The accuracy of a signature is directly proportional to the length of the call stack suffixes. A deadlock bug can have multiple signatures, each of them corresponding to a different manifestation of the deadlock.

Dimmunix consists of two components: (1) an avoidance module that prevents reoccurrences of previously encountered deadlocks, and (2) a detection module that detects deadlocks, extracts their signatures, and adds them to a persistent history. Dimmunix requires no assistance from programmers or users, and can be used by users to defend against deadlocks while waiting for a vendor patch. Dimmunix runs within the address space of the target program.

Before each lock acquisition, the avoidance module decides whether to allow the running thread to proceed with the lock acquisition. Avoiding deadlocks requires anticipating whether the lock acquisition would lead to the instantiation of a signature from the deadlock history. For a signature with outer call stacks CS_1, ..., CS_n to be instantiated, there must exist threads $t_1,\ldots,t_n$ that either hold or are block waiting for locks $l_1,\ldots,l_n$ while having call stacks $CS_1,\ldots,CS_n$. If no signature from the deadlock history can be instantiated, the avoidance module allows the caller thread to proceed with the lock acquisition; otherwise, it suspends the thread until the lock acquisition cannot cause any instantiation of a signature from the history.

B. Collaborative Deadlock Immunity

In collaborative deadlock immunity, different machines connected to the Internet work together to achieve immunity against deadlocks by sharing their deadlock signatures.

An important benefit of sharing the deadlock signatures is that any application can use the collective knowledge of the other nodes to generalize deadlock signatures from its history. The generalization consists of merging different signatures of the same deadlock bug. The role of signature generalization is to keep few signatures per deadlock bug, in order to have a small size of the deadlock history for each application. If all possible manifestations of a deadlock bug D were experienced by some nodes in the Internet, the current signatures of D are the most accurate signatures that enable Dimmunix to avoid all the manifestations of D.

Upon receiving a signature from other nodes, Communix checks whether the signature can be used by the running application. This first validation step assumes the node that sent the signature is honest.

Attackers may send fake deadlock signatures that do not represent real deadlock bugs; these signatures may cause denial of service (DoS) in applications instrumented with Dimmunix. Such signatures may exploit Dimmunix to increase the runtime overhead of signature matching and reduce the parallelism due to suspending threads. The validation process should prevent such signatures from harming the performance or the functionality of the applications. Therefore, additional checks are performed (§III-C).

III. Design

In this section, we describe the architecture of the Communix framework (§III-A), explain the signature distribution (§III-B), and describe in detail the signature validation (§III-C) and signature generalization (§III-D).

A. Communix Framework

Communix has five components, as we illustrate in Figure 1: Dimmunix (i.e., the deadlock immunization component of Communix), Communix plugin, Communix server, Communix client, and Communix agent. Dimmunix is in charge of (1) detecting deadlocks, (2) saving their corresponding signatures into the running application's deadlock history, and (3) preventing the application from encountering the same deadlocks again.

Communix uses a centralized signature distribution framework. The Communix plugin, implemented on top of

Dimmunix, sends the deadlock signatures to the Communix server, right after Dimmunix produces the signatures. In order to obtain new deadlock signatures from the server, a machine must have the Communix client installed; the client periodically downloads the new deadlock signatures from the server into a local repository. Any Java application running with Dimmunix can use these signatures to improve its protection against deadlocks.

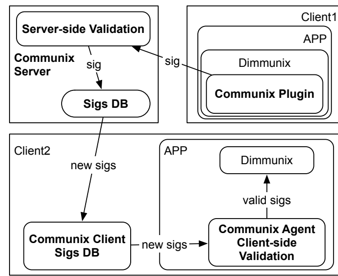

Figure 1. Communix architecture.

A centralized signature distribution improves the protection against deadlocks for all the machines connected to the Internet that are equipped with Communix. Each newly discovered signature S becomes available to any machine connected to the Internet; as soon as other nodes download S from the Communix server and validate it, they are protected against deadlock manifestations matching S, without having to encounter S.

The client-side signature validation and signature generalization are performed by the Communix agent. The agent runs together with Dimmunix, in a Java application's address space. When the application starts, the agent selects from the local repository the new signatures that are valid, i.e., that can be used by the application. If a new signature S is found valid, the agent attempts to merge S with an existing signature from the running application's deadlock history. If S cannot be merged with any existing signature, then it represents a new deadlock bug; the agent adds S to the history, in order to prevent future occurrences of deadlocks matching S.

To validate a new deadlock signature, the Communix agent checks whether the signature matches the running application. In addition, Communix protects the users against DoS attacks based on distributing malicious signatures. To generalize deadlock signatures, Communix merges signatures representing the same deadlock bug into one signature.

This paper focuses on deadlock bugs; however, a similar collaborative immunity framework can be imagined for other bugs, like data races and atomicity violations.

B. Signature Distribution

Communix allows different users running the same application (or different applications sharing some deadlock-prone library) to share signatures. The more users run some deadlock-prone code, the more likely it is that all possible manifestations of the deadlock bug are experienced in a short period of time, and all users get fully immunized against the deadlock.

Once Dimmunix detects a deadlock, the Communix plugin sends the corresponding signature to the Communix server. The Communix server collects in a database all the deadlock signatures discovered by Java applications running with Dimmunix on arbitrary machines connected to the Internet. To decide whether to add an incoming signature to the database, the server performs a simple signature validation, described in §III-C2.

The Communix client, running on an arbitrary machine in the Internet, periodically downloads the new deadlock signatures from the server into a local repository. The local repository is updated once a day; a high frequency (e.g., once a minute) would overload the Communix server. The updates are incremental, i.e., the client requests from the server only the signatures that are not present in the local repository. When a Java application A starts, the Communix agent inspects the new signatures from the local repository: the agent checks the validity of each new signature S (§III-C3); if S is valid, the agent adds S to A's deadlock history. The inspection of the local repository is incremental, i.e., every signature is analyzed only once.

The Communix client runs as a background process, decoupled from the agent. Without this decoupling, the Communix agent would have to connect to the server and retrieve new deadlock signatures every time a Java application starts. This would introduce an unnecessary overhead.

Note that Communix does not require users to provide any application specific information (like name or version) with the signatures they share. Communix only needs hash values of class bytecodes, in order to distinguish different versions of the same class or different classes having the same name. The hash values are automatically computed by the Communix plugin, when Dimmunix produces the signatures. This makes Communix application agnostic.

C. Signature Validation

Before sending a signature to the server, the Communix plugin attaches to each call stack frame of the signature the hash of the class bytecode containing that frame.

Each time a Java application running Dimmunix starts, the Communix agent selects from the local repository the signatures that match the running application. The agent checks whether the hashes of an incoming signature match the bytecode hashes of the running application. If the hashes do not match for all the top frames, the signature is rejected; otherwise, the agent keeps from the signature the longest call stack suffixes with hashes matching the application.

If all nodes were honest, the above check would have been sufficient; unfortunately, there are attackers that may try to exploit Dimmunix by sending fake signatures, therefore additional checks are needed.

1) Preventing (Containing) DoS Attacks: An attacker may attempt to perform a performance DoS attack based on signature flooding, to put pressure on Dimmunix's signature matching mechanism. Such an attack consists of sending many fake signatures that manage to pass the validation and get accepted into the deadlock history of an application.

This would put pressure on Dimmunix, because all these signatures have to be matched at runtime.

Communix manages to prevent such attacks by performing three additional checks. The first two checks are performed by the server, and the third one is performed by the Communix agent, on the client side. If any of these checks fails, the signature is rejected.

First, the server requires each incoming signature to be accompanied by an encrypted id of the sender. The encrypted id is provided once by the Communix server. The server uses the sender ids to bind each incoming signature to the user who sent the signature. Since an attacker can fake many IP addresses, they cannot be used to identify the senders; it must be hard for an attacker to obtain multiple ids.

Second, the server makes sure that every two distinct signatures sent by the same user (i.e., having the same sender id) have no common top frames. This restriction should not affect honest users, because it is not likely that a user would experience such "adjacent" deadlocks. However, if he/she does experience such situations, the signatures wrongly rejected due to this restriction can be provided by other users.

Third, the agent checks whether the outer call stacks of a new signature end in nested synchronized blocks/methods. Checking whether a synchronized block/method is nested is straightforward, due to the disciplined way the Java compiler nests these constructs. We describe the algorithm in §III-C3. Communix does not handle explicit lock/unlock operations (e.g., calls to *ReentrantLock.lock/unlock()*). However, this is a minor deficiency, since Java programs use mostly synchronized blocks/methods (§IV).

Thanks to the above three checks, the possibility to flood Dimmunix with fake signatures is limited. If there are N nested synchronized blocks/methods in a Java application A, an attacker cannot "provide" more than N signatures that get accepted into A's deadlock history. Typically, in a Java application there are a few hundred nested synchronized blocks/methods (§IV). Therefore, an attacker cannot force more than a few hundred signatures into the deadlock history of an application.

Another type of performance DoS attack that an attacker may attempt is to send fake signatures that slow down an application. These attacks force Dimmunix to avoid instantiations of fake signatures or signatures that are too general. The more general a signature is, the more often Dimmunix has to avoid instantiations of the signature. This means Dimmunix suspends threads more often then needed, which may considerably slow down the application. The attacker may exploit the generalization mechanism to retain only the top frames of the outer call stacks, or send directly signatures with outer call stacks of depth 1.

The Communix agent prevents the attackers from sending signatures with outer call stacks of depth < 5. For the applications we studied, the outer call stacks have large depths (usually > 10); therefore, we believe that this restriction does not affect the honest users. Signatures with outer call stacks of depth 5 incur an acceptable performance overhead; for depth 1, the overhead is considerable (§IV-B). Therefore, the outer call stacks must have the depth ≥ 5. To prevent an attacker from exploiting the signature generalization mechanism to obtain outer call stacks of depth 1, the agent does not merge signatures below depth 5, for the outer call stacks. Alternatively, one could compute the minimal depth d that outer call stacks corresponding to a nested synchronized block/method can have; the threshold would be $min(d,5)$, rather than 5, in this case.

The third check ensures that the worst damage an attacker can do is to force into the deadlock history of an application signatures with outer call stacks of depth 5, that cover all the nested synchronized blocks/methods. We show in §IV that such a scenario causes only 8-40% performance overhead in the Java applications we studied.

An attacker may also attempt a functionality DoS attack that disables features of an application. If a certain feature needs some code to execute concurrently, that feature would no longer be available, if Dimmunix makes the code execute sequentially. This undesired effect can be caused also by real deadlock signatures; some concurrent code may be deadlock-prone, and execute most of the time without deadlocking. To prevent such situations, we use Dimmunix' false positive detection mechanism. If after 100 instantiations of a signature S there was no true positive, and there was at least one interval of 1 second having more than 10 instantiations of S, Dimmunix decides to warn the user about signature S; the user can decide to keep S, if he/she notices no change in the behavior of the application.

Attackers may attempt to put pressure on the Communix server by sending bursts of fake signatures to the server. The server processes only up to 10 signatures per day from one user; beyond this threshold, the signatures from that user are ignored by the server. This restriction usually does not affect honest users, since it is unlikely that a user would experience so many different deadlocks (or different manifestations of a deadlock) in 1 day. However, the wrongly rejected signatures can be provided by other users.

In the remainder of this section, we describe in detail the server-side and client-side signature validation.

2) Server-side Signature Validation: The Communix server requires each user to accompany the signatures he/she sends with an encrypted user id that the server provides. The server provides a unique id to each user; the id is encrypted, in order to prevent users from manufacturing their own ids. To be able to share its signatures, each user has to previously obtain the encrypted id from the Communix server. The server uses AES encryption, with a predefined 128-bit key, to produce the encrypted user ids. We did not implement the service for issuing the encrypted user ids; such a service exceeds the scope of this work. The problem of preventing attackers from impersonating multiple users

has been extensively studied.

Upon receiving a signature S accompanied by an encrypted id I, the Communix server decrypts I to obtain the id of the user that sent the signature. After retrieving the sender id, the server checks whether the same user already sent a signature S' which is adjacent to S, i.e., S and S' have some (but not all) top frames in common. If the user already sent a signature adjacent to S, the server refuses to add S to its database.

Rejecting adjacent signatures from the same user considerably reduces the capability of an attacker to provide fake signatures. Assume there are N synchronized blocks (methods) in an application, and there are N_d possible call stack suffixes of depth d, for each synchronized block (method). Without this restriction, the attacker can manufacture $(N \cdot N_d)^4$ signatures of two-thread deadlocks that pass the validation, for each depth $d \geq 5$; this gives a total of $N^4 \cdot \sum_{d=5}^{\infty} N_d^4$ possible signatures. With this restriction, the attacker can provide only N signatures.

3) Client-side Signature Validation: For each new signature S, the Communix agent checks whether S matches the running Java application, then it checks whether the outer call stacks of S end in nested synchronized blocks/methods.

For each call stack of signature S, the Communix agent checks whether the hashes it carries match the running application A. Each call stack of signature S is encoded as a sequence of frames $[c_1.m_1 : l_1 : h_1, ..., c_n.m_n : l_n : h_n]$, where c_i are class names, m_i are method names, l_i are line numbers, and h_i is the hash of class c_i's bytecode. The hashes are attached by the Communix plugin when Dimmunix produces signature S. The hash value h_k matches application A if and only if class c_k's bytecode from application A has the hash h_k. The hash check starts from the top frame, i.e., frame n; if h_n does not match A, signature S is rejected. If h_k $(1 \leq k < n)$ is the first hash value that does not match A, the frames $1, ..., k$ are removed from the call stack; if all hashes match, the call stack remains unchanged. For efficiency, the Communix agent computes the hash of a class first time the class is loaded, then it reuses the computed hash value.

The hash checking covers also the inner call stacks, even though they are not used by Dimmunix for deadlock avoidance. The signature may correspond to an earlier version of the application, where the code between the outer and inner lock statements was deadlock-prone. That code might have been fixed in a newer version of the application. If the Communix agent would check only the hashes of the outer call stacks, these code changes would be missed, and the false positive signature would pass the validation.

We describe now the algorithm for checking whether a synchronized block B is nested. Given the control flow graph (CFG) of an application binary, and the *monitorenter* statement s corresponding to a synchronized block, the Communix agent inspects the CFG, starting from the successor of s. As soon as a *monitorenter* (*monitorexit*) statement is encountered, the algorithm returns that B is nested (non-nested). If a method call statement s_{call} is met, the algorithm returns that B is nested, if any method that may be called (directly on indirectly) by s_{call} is either synchronized or contains a synchronized block.

Since a synchronized method is semantically equivalent to a *synchronized(this)* block that wraps the method body, the algorithm for checking whether synchronized methods are nested is similar. In fact, the AspectJ instrumentation framework [2] that Dimmunix uses transforms the synchronized methods into synchronized blocks.

For efficiency, the Communix agent precomputes the locations of all the nested synchronized blocks/methods, when the application runs for the first time. Checking if the outer call stacks of a signature end in nested synchronized blocks/methods consists of determining if the top frames belong to this precomputed set of locations. To inspect the application bytecode, the Communix agent uses the Soot bytecode analysis framework [3].

Each time new classes are loaded, in addition to the ones loaded in the previous runs, the Communix agent repeats the nesting check for all the signatures from the local repository that passed the hash check and failed the nesting check. There is no need to recheck the nesting for the rest of the signatures, because adding new classes to the CFG can only uncover new nested synchronized blocks/methods.

D. Signature Generalization

The signature generalization consists of merging different signatures corresponding to the same deadlock bug, i.e., that end in the same inner and outer lock statements. The resulting signature consists of the longest common suffixes of the call stacks forming these signatures.

It is important to generalize signatures for the following reason. If the outer call stack suffixes are long, the signature may not be able to always avoid the deadlock. In other words, there may be false negatives, i.e., other signatures of the same deadlock ending in different outer call stack suffixes. If there are multiple manifestations of the deadlock having different outer call stack suffixes, it may take a long time until a single user experiences all these manifestations.

A trivial solution to avoid all the possible manifestations of a deadlock would be to match only the top frames of the signature's outer call stacks. However, there is an important drawback to this solution: having the outer call stacks matched too shallowly introduces false positives [1] and therefore reduces the parallelism, which may have a negative impact on performance.

The generalization process is the following: When a Java application starts, the Communix agent checks if new signatures that passed the validation could be merged with existing signatures from the deadlock history of the running application. The signatures that cannot be merged are added to the history.

Two signatures S and S' can be merged if and only if they represent the same deadlock bug (i.e., the top frames of S have to be identical to the top frames of S'), and either (1) S and S' were produced on the local machine, or (2) S/S' is a remote signature and the resulting signature has the outer call stacks of depths ≥ 5.

Merging two signatures consists of finding the longest common call stack suffixes of the two signatures. Given two signatures $S = \{(CS_{1,out}, CS_{1,in}), ...\}$ and $S' = \{(CS'_{1,out}, CS'_{1,in}), ...\}$, their generalization is the signature $S^g = \{(CS^g_{1,out}, CS^g_{1,in}), ...\}$, where CS^g is the longest common suffix of call stacks CS and CS'.

IV. Evaluation

In this section, we first evaluate the performance of Communix (§IV-A), then we evaluate the impact DoS attacks can have on Java applications running Dimmunix (§IV-B). Finally, we estimate the time it takes for an application to achieve full deadlock protection with Communix, compared to using Dimmunix alone (§IV-C).

The experiments were run on machines with two 4-core Intel Xeon 2GHz processors each, 20 GB of memory, running Ubuntu Linux 10.04.

A. Communix's Performance

In this section, we first evaluate the performance of the Communix server, then the performance of the whole signature distribution in an end-to-end setting. Then, we evaluate the performance of the client-side signature validation plus the signature generalization. Since the signature generalization and client-side signature validation are both performed by the Communix agent at application startup, we decided to evaluate them together. Finally, we measure the time it takes for the Communix agent to find the nested synchronized blocks/methods; the time it takes to compute the hashes of the loaded classes is negligible compared to the time it takes to perform the nesting analysis.

The server processes two types of requests: an *ADD(sig)* request that means "add signature *sig* to the database", and a *GET(k)* request that means "send me the signatures from the database starting from index *k*". Normally, a client having a local repository with *n* signatures sends *GET(n+1)* requests to the server to retrieve the new signatures. We wanted to evaluate worst case scenarios, therefore we use only *GET(0)* requests in our measurements, which means that the server is always asked to send all its signatures.

To evaluate the server's performance, we invoke the request processing routines from 1,000-100,000 simultaneous threads. This test measures the efficiency of the server's computations, i.e., adding new random signatures to the database (including the server-side signature validation) and iterating through the entire database. Figure 2 shows that the server scales well up to 30,000 simultaneous

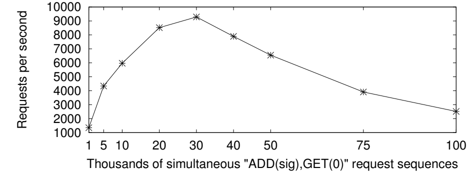

Figure 2. The performance of the Communix server.

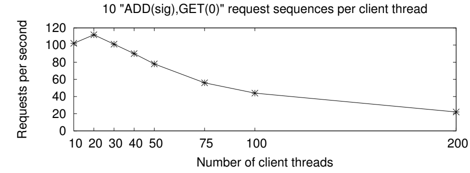

Figure 3. The performance of the signature distribution.

"*ADD(sig),GET(0)*" sequences of requests. At its peak, the server processes 9,000 requests per second.

We evaluate the performance of the signature distribution in an end-to-end setting. On one machine we ran the Communix server, and on another machine we ran 10-200 client threads that send 10 "*ADD(sig),GET(0)*" sequences of requests each. Figure 3 shows that the signature distribution scales well up to 30 client threads, i.e., 300 simultaneous "*ADD(sig),GET(0)*" sequences of requests. However, the throughput (i.e., requests served per second) is up to two orders of magnitude lower compared to Figure 2. The explanation is that the network communication between the server and the client threads becomes a bottleneck. The size of a signature is 1.7 KB. If there are N client threads and each thread sent on average k "*ADD(sig),GET(0)*" request sequences to the server, the server has to send $(k+1/2) \times N^2 \times 1.7$ KB of data to the N clients, on average, to serve the next round of *GET(0)* requests. If $N = 200$, the server has to send in the 10th round approximately 630 MB of data to the 200 clients. To summarize, a server with one network card cannot distribute signatures fast if multiple clients ask simultaneously for a large number of signatures.

As shown in Figure 3, a client thread receives 20-110 replies per second to "*ADD(sig),GET(0)*" request sequences, from the Communix server. Therefore, it takes 9-50 milliseconds to send the two requests to the server and get the replies. However, the latency of the signature distribution is up to 1 day, because the Communix client downloads the new signatures from the Communix server only once a day.

We evaluate the Communix agent on large Java applications, i.e., JBoss, Limewire, and Vuze. JBoss is a well-known Java application server, while Limewire and Vuze are well-known peer-to-peer file sharing applications. For each application, we measure the time it takes to start and immediately shut down. In Figure 4, we show the performance of the computations performed at startup by the agent, i.e., client-side signature validation and signature generalization. For up to 1,000 new signatures in the local repository, the Communix agent incurs a startup delay of up to 2-3 seconds, i.e., 11-16% startup slowdown.

In Table I, we show the efficiency of the static detection

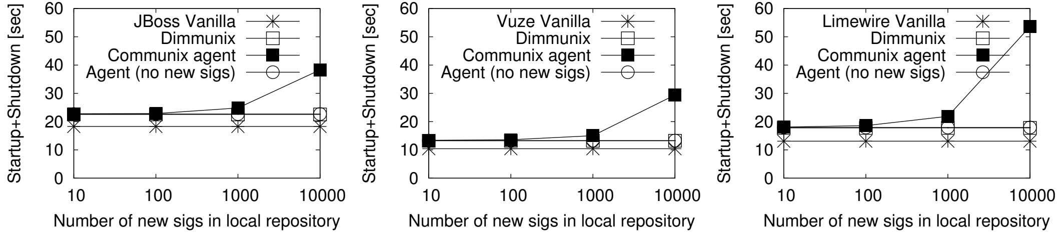

Figure 4. The performance of client-side computations, i.e., client-side signature validation and signature generalization.

Table I
STATISTICS ABOUT VARIOUS JAVA APPLICATIONS, AND THE PERFORMANCE OF THE NESTING ANALYSIS.

App	Size(LOC)	Sync bl/meths	Explicit sync ops	Nested (Analyzed)	Nesting check(sec)
JBoss	636,895	1,898	104	249 (844)	114
Limewire	595,623	1,435	189	277 (781)	122
Vuze	476,702	3,653	14	120 (432)	50

of nested synchronized blocks/methods and some statistics we collected about the three applications, i.e., size in lines of code (LOC), number of synchronized blocks/methods, number of explicit lock/unlock operations (i.e., calls to *ReentrantLock.lock/unlock()*), and the number of nested synchronized blocks/methods that the nesting analysis reports. The Communix agent could analyze only 11-54% of the synchronized blocks/methods. For the rest of the synchronized blocks/methods, the Soot static analysis framework could not retrieve enough information for the nesting analysis (i.e., it could not retrieve the CFGs of some of the methods). Table I shows that it takes 50-122 seconds to analyze 432-844 synchronized blocks/methods. The nesting analysis is performed at shutdown, first time the application runs, and each time new classes (w.r.t. the previous run) are loaded. Therefore, the analysis is performed only for the first couple of runs. Moreover, since the analysis is performed at shutdown, the delay is not bothersome for the user, if the user does not intend to restart the application soon.

B. The Impact of DoS Attacks

The attackers have only one way to exploit Dimmunix, to slow down a Java application: they can send signatures with outer call stacks of depth 5 which cover all the nested synchronized blocks/methods that are on the critical path, in order to maximize the amount of thread serialization in applications running Dimmunix. If there is already a signature S in the deadlock history that can be merged with a malicious signature S', signature S' will replace S in the history, by exploiting the generalization mechanism.

Table II shows that attackers providing malicious deadlock signatures can cause only 8-40% performance overhead in the studied real applications running with Dimmunix. The tests run with 20 deadlock signatures in the history, with outer call stacks of depth 5. These outer calls are on the critical path, i.e., more than 99% of the nested synchronized blocks/methods are executed with these call stacks. In this worst case scenario, the performance overhead incurred by Dimmunix is 8-40%, which is acceptable for general-purpose applications. If none of the signatures is on the critical path, the performance overhead incurred by Dimmunix is negligible (i.e., $< 2\%$). For outer call stacks of depth 1, the performance overhead is considerable (i.e., $> 100\%$), for some of the applications we studied. However, this situation is avoided, because the Communix agent does not accept incoming signatures with outer call stacks of depth < 5. Therefore, Communix successfully contains DoS attacks.

Table II
WORST CASE OVERHEAD INCURRED WHILE UNDER A DOS ATTACK.

Application	Benchmark/Test	Overhead
JBoss	RUBiS	40%
MySQL JDBC	JDBCBench	38%
Eclipse	Startup + Shutdown	33%
Limewire	Upload test	10%
Vuze	Startup + Shutdown	8%

Making it hard for a user to obtain multiple encrypted ids from the Communix server, together with restricting the server to process only up to 10 signatures per day for the same user id, protects the server and the clients against flooding with fake signatures. Assuming 100 attackers manage to obtain 5 ids each from the server, and they keep sending fake signatures to the server, the attackers could make the server process and add to its database only up to $100 * 5 * 10 = 5,000$ signatures in 1 day. Assuming the worst case, i.e., the 5,000 signatures are sent simultaneously by the 100 attackers, the server can process the signatures in 1 second, the Communix client can download them in a few minutes, and the agent can process them in 10-15 seconds.

C. Time to Achieve Full Protection Against Deadlocks

As we mentioned in §III-D, it may take a long time for a single user to experience all the deadlocks of an application and all the manifestations of these deadlocks. Therefore, it may take a long time until Dimmunix alone can provide full protection against deadlocks.

If there are many users of an application A, Communix can considerably reduce the time it takes for A to be deadlock-free. The time it takes for Communix to provide full protection against deadlocks for application A is inversely proportional to the number N_u of users that run A in different ways. If there are N_d possible deadlock manifestations in A and it takes on average t days for a user to experience one manifestation, A will be deadlock-free in roughly $t * N_d$ days, if Dimmunix alone is used. If Communix is used, all the users of A will have A deadlock-free in roughly $t * N_d / N_u$ days. The larger N_u, the higher the

gain that Communix brings.

The estimate we made here is purely theoretical. A real evaluation is possible only if Communix is deployed in the field and statistics are collected after a considerable period of usage (e.g., months) from many (e.g., thousands) users.

V. Related Work

In this section, we review the literature on approaches to avoiding application failures.

Static analysis tools look for bugs at compile time, helping programmers remove them. ESC [4] uses a theorem prover and relies on knowledge from annotations generated by Houdini [5]. Larochelle et al. [6] presents an approach to detect vulnerabilities through a source code analysis. Relay [7] and KLEE [8] use symbolic execution to statically detect bugs in applications; however, exponential growth of execution paths limits scalability of the symbolic execution. RacerX [9] provides a static flow-sensitive, interprocedural analysis to detect deadlocks. Static analysis tools can run fast, avoid runtime overhead, and help prevent deadlocks. However, they produce false positives, and it is ultimately the programmers' burden to find the true positives.

Dynamic bug detection is conceptually observing the program execution to extract various kinds of information. Nir-Buchbinder et al. [10] dynamically discover deadlocks and instrument the code using a "gate lock" to prevent similar deadlocks in the future. [11] serializes threads' access to lock sets that could induce deadlocks. GoodLock [12], [13] detects deadlocks by recording the nested locking pattern for each thread. Rx [14] rolls back to a checkpoint upon deadlock and retries the execution in a modified environment. Machine learning techniques are also used for dynamic bug detection. ClearView [15] instruments the applications to dynamically profile the execution flows. ClearView employs the profiles to distinguish erroneous executions later.

Fingerprinting the anomalous execution signatures enables applications to avoid reoccurrences of bugs in the future. Bouncer [16], using DFI [17] for dynamic bug detection, generates signatures to block exploits before they get processed by the program. Snort [18] provides a signature-based exploit detection, saving attack signatures once a new attack is detected. The manual generation of the signatures limits Snort's scalability.

Finally, there exist approaches addressing cooperative security (dependability). In particular, once a bad execution pattern is detected and its corresponding signature is generated, the application helps its peers by broadcasting the signature to its neighbors. Vigilante [19] proposes an end-to-end approach to contain worms automatically. It relies on collaborative worm detection at end hosts, but does not require hosts to trust each other. Communix differs from Vigilante in the validation of the received bug signatures: Vigilante uses replay to validate a new signature, while Communix efficiently checks deadlock signatures statically.

VI. Conclusion

We have presented Communix, a collaborative deadlock immunity framework that targets deadlock bugs in general-purpose Java applications. Communix complements Dimmunix [1]. Dimmunix fingerprints the execution flows leading to deadlocks, then Communix sends these fingerprints (signatures) to a deadlock immunity server that makes them available to all nodes in the Internet. A Communix agent running within Dimmunix selects the new signatures that match the running Java application. The accepted signatures are stored and used to prevent deadlocks. An important contribution of Communix is that it uses the collective knowledge of all nodes in the Internet running the same application, to improve the protection against deadlocks for each individual node running that application. Communix is efficient and scalable: the server can process up to 9,000 requests per second, and the agent can validate 1,000 new signatures in 2-3 seconds, while managing to contain DoS attacks. This makes Communix an attractive framework for providing collaborative immunity against deadlock bugs.

References

[1] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Symp. on Operating Systems Design and Implementation*, 2008.

[2] "AspectJ," http://www.eclipse.org/aspectj.

[3] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co, "Soot - A Java optimization framework," in *Conf. of the Centre for Advanced Studies on Collaborative Research*, 1999.

[4] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Conf. on Programming Language Design and Implementation*, 2002.

[5] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *Intl. Symp. on Formal Methods Europe*, 2001.

[6] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, 2001.

[7] J. W. Voung, R. Jhala, and S. Lerner, "Relay: Static race detection on millions of lines of code," in *Symp. on the Foundations of Software Eng.*, 2007.

[8] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symp. on Operating Systems Design and Implementation*, 2008.

[9] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *Symp. on Operating Systems Principles*, 2003.

[10] Y. Nir-Buchbinder, R. Tzoref, and S. Ur, "Deadlocks: From exhibiting to healing," in *Workshop on Runtime Verification*, 2008.

[11] F. Zeng and R. P. Martin, "Ghost locks: Deadlock prevention for Java," in *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[12] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, 2000.

[13] L. W. Agarwal and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, 2005.

[14] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, "Rx: Treating bugs as allergies – a safe method to survive software failures," *ACM Transactions on Computer Systems*, vol. 25, no. 3, 2007.

[15] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *SOSP*, 2009.

[16] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: Securing software by blocking bad input," in *Symp. on Operating Systems Principles*, 2007.

[17] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Symp. on Operating Systems Design and Implementation*, 2006.

[18] M. Roesch, "Snort - lightweight intrusion detection for networks," in *LISA '99: Proceedings of the 13th USENIX conference on System administration*, 1999.

[19] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of Internet worms," in *Symp. on Operating Systems Principles*, 2005.