# Cross Application Data Provenance and Policy Enforcement

BRIAN DEMSKY
University of California, Irvine

We present a new technique that can trace data provenance and enforce data access policies across multiple applications and machines. We have developed Garm, a tool that uses binary rewriting to implement this technique on arbitrary binaries. Users can use Garm to attach access policies to data and Garm enforces the policy on all accesses to the data (and any derived data) across all applications and executions. Garm uses static analysis to generate optimized instrumentation that traces the provenance of an application's state and the policies that apply to this state. Garm monitors the interactions of the application with the underlying operating system to enforce policies. Conceptually, Garm combines trusted computing support from the underlying operating system with a stream cipher to ensure that data protected by an access policy cannot be accessed outside of Garm's policy enforcement mechanisms. We have evaluated Garm with several common Linux applications. We found that Garm can successfully trace the provenance of data across executions of multiple applications and enforce data access policies on the application's executions.

Categories and Subject Descriptors: []:

General Terms:

Additional Key Words and Phrases:

## 1. INTRODUCTION

Most computing systems do not store information about the history (provenance) of data. Users cannot query about the information sources or the chain of people and programs that contributed to the creation of data. The lack of information about data provenance profoundly affects computing. With today's systems, it can be very difficult to audit whether a file contains information that was appropriated from potentially untrusted sources. Data provenance can also serve a useful role in recovering from known security issues. For example, a recent software bug in the Debian ssh potentially causes cryptographically weak keys to be generated. In such situations, data provenance could help determine which keys are affected.

Releasing information on current systems means losing control over it, including who has access to it and what they do with it. For example, providing personal information to any entity incurs the risk of accidental release on the Internet. Indeed, numerous news stories document accidental releases of the personal information of millions by companies and government agencies. There have been limited efforts to create systems that enforce policies on how data is used — the entertainment industries have created digital right management (DRM) systems with the primary goal of preventing protected media files from being pirated. Current systems impose draconian limitations on how consumers use the media files — they often only support playing the protected media. Moreover, these systems typically restrict consumers to a very limited set of applications. The reason for these restrictions is

that once protected data leaves the small number of trusted applications, there is no mechanism to continue enforcing the data policies.

This paper presents a tool, Garm, that provides a mechanism to both trace the provenance of the data applications process and enforce access policies on this data. We introduce a new approach to security policies that make them straightforward to use in current computing environments. Unlike previous work, our data access policies follow the protected data across execution, application, and machine boundaries.

Previous work on information flow required developers to specify the locations where trusted data can be stored and then used policies to ensure that an application could not leak protected data to an untrusted location. One problem with such approaches is that in order to enforce policies that they must impose a rigid structure on where users can store their data and how they can share it even for legitimate uses. Such rigid security models are likely to be problematic in real work environments — they deny users the flexibility they need to perform legitimate work. For example, such mechanisms make it difficult for employees to legitimately share protected data between computers through mechanisms like USB drives. Garm provides a much more flexible security model — applications can write policy-protected data to any location including untrusted ones without losing policy enforcement. Garm traces the policies that apply to a program's output files through the use of provenance shadow files and uses fine-grained encryption to prevent outside accesses to the data that circumvent the policy enforcement mechanism.

Another issue with previous approaches is that they are often not amenable to composing policies that operate on orthogonal data classification systems. Because these systems only explicitly trace classifications within a single application, the choice of location to store data conceptually encodes the classification information for the data. Supporting multiple orthogonal classification systems would conceptually require creating storage locations for each possible combination of classifications. Our approach trivially handles composing different policies and data classification systems.

Garm enables off-the-shelf binaries to process protected data while a dynamic analysis traces both the provenance of the data inside the application and which policies apply to the data. The rewriting system is responsible for enforcing the policies, thereby allowing policy-protected data to be used with existing applications while still enforcing the policies. The ability to enforce policies across a wide range of applications provides multiple potential benefits ranging from the security of personal data to new, more user-friendly and flexible digital rights management.

### 1.1 Basic Approach

Garm creates provenance records that characterize the set of program executions and information sources that have contributed or modified the data over its lifespan. It uses binary rewriting to instrument binaries with code that traces the provenance of an application's state during its execution. The binary rewriter uses a static provenance analysis to generate an optimized dynamic instrumentation. When an application accesses new data, Garm creates a base provenance record that describes the source of the data. If Garm had previously monitored the program execution

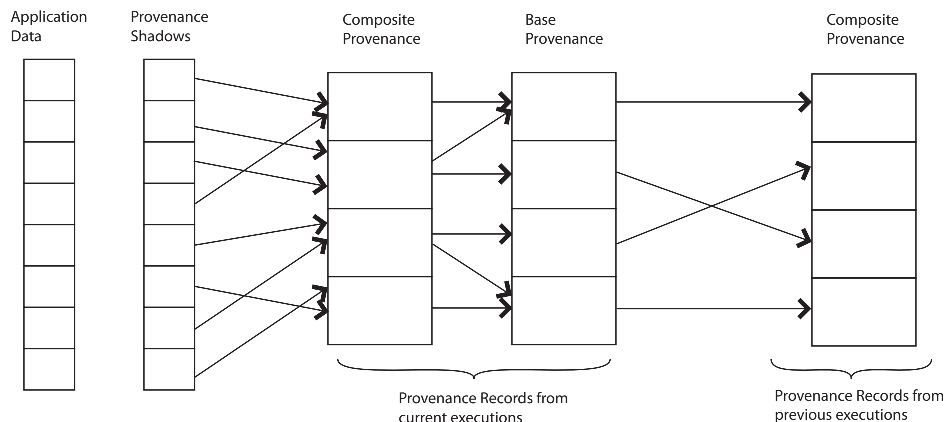


Fig. 1. Provenance Records

that produced the data, this base provenance will reference the provenance record from that previous execution. When the monitored program performs operations that depend on data with the different provenances, Garm labels the bytes produced by the operation with a composite provenance that lists the set of base provenances that contribute to the current value. In this fashion, the dynamic analysis computes shadow composite provenances for all bytes that the application writes.

Figure 1 presents the structure of Garm's provenance records. When an application writes data to a file, Garm generates a shadow file that stores the provenances of the bytes in the file. Garm also stores a provenance record for each execution that describes the source for each base provenance and lists the base provenances that contribute to each composite provenance. If shadow files are transferred with (or bundled with) files when transferring files between machines, Garm can trace provenance across machines. Note that if a user does not transfer a shadow file, it is impossible to access policy-protected data. We assume the presence of a shared server to store provenance records for executions. If a shared server is undesirable, copies of all composite provenance records referenced by the shadow file can be packed at the end of the shadow file.

Garm supports data access policies by allowing users to attach data access policies to provenances. Before allowing an application to output data, Garm checks whether the output operation is permitted by the data's access policies. The access policies can either (1) allow outputting unprotected data to the location (i.e. audio device, screen, etc), (2) allow outputting encrypted data along with the policy and provenance information, or (3) prohibit outputting the data in any form to the location. The ability to output encrypted data and policy information enables users to flexibly share and manipulate data in all the ways they currently do without losing policy protection. The policy can depend on context (i.e. the date, how many times the data has been accessed, etc). Garm's fine-grained encryption based on stream ciphers enables users to seamlessly share arbitrary files that contain policy-protected data between applications while ensuring that programs cannot use policy-protected data in ways that violate the access policy.

### 1.2 Contributions

This paper makes the following contributions:

- **Data Protection Framework:** It introduces a new data protection framework. This framework encrypts policy-protected data before it is passed to the operating system and decrypts policy-protected data before an authorized application reads it. Garm uses the shadow provenance file to record which policy server holds the keys to access the file. This mechanism allows Garm protected-data to be secure even when stored in untrusted locations.
- **Data Policy Enforcement:** It presents a generic mechanism that enforces data access policies on arbitrary binary applications.
- **Data Provenance Analysis:** It presents an analysis that can trace the provenance of an application's state. The analysis combines static and dynamic analyses together to determine which information sources were used to derive each value in the execution.
- **Cross Application Support:** It presents a new runtime mechanism that uses stream ciphers together with provenance shadow files to prevent data accesses that circumvent Garm's policy enforcement. Garm introduces support for tracing provenance information across executions and application boundaries.

The remainder of the paper is structured as follows. Section 2 presents Garm's architecture. Section 3 presents an overview of the provenance analysis. Section 4 describes limitations of the approach. Section 5 presents our experience using Garm. Section 6 presents related work; we conclude in Section 7.

## 2. SYSTEM OVERVIEW

We next describe Garm's architecture. Figure 2 presents an overview of Garm. The system consists of a binary rewriting engine that traces provenances and enforce policies, a set of policy servers that manage policy keys, provenance records that describe how a given provenance record was derived from previous provenance records and new input files, and provenance shadow files that record the provenance of the data stored in files.

The rewriting system intercepts the system calls that the guest application performs. When the guest application loads data from a file into an address in the application's memory, the binary rewriting engine loads the provenance from the corresponding provenance shadow file into the provenance shadow for that address. We have designed Garm to store provenance and policy information in a separate shadow file to allow applications outside of Garm to read non policy-protected data from the files.

If the provenance indicates that the data is protected by a policy, the system sends the encrypted policy information to the policy server along with information about the application and the user's identity. Garm would use the remote attestation capability that is commonly provided by trusted platforms to enable the policy server to verify that neither Garm nor the underlying components have been tampered with [Mitchell 2005]. We intend that policy servers will be hosted by the individuals who create policies or their agents. The policy server verifies that neither Garm nor the underlying components have been tampered with by verifying the remote attestation as describe in Section 2.2. Then the policy server decrypts

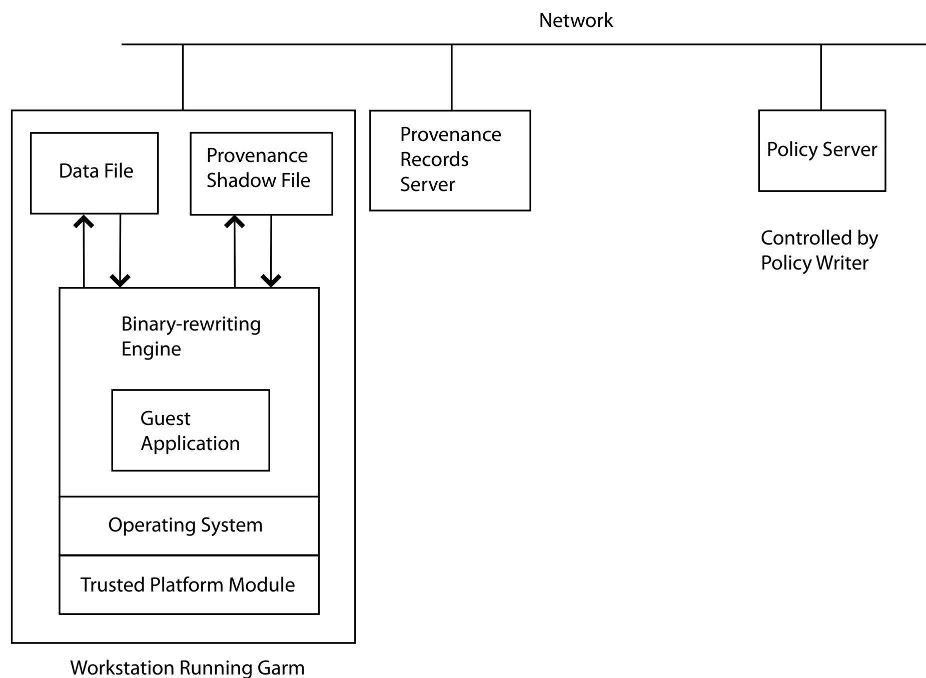


Fig. 2. System Overview

the policy information to obtain the policy and the policy keys. The policy server checks the policy against the information from Garm. If no policy violations are detected, the policy server sends the policy along with the policy key to Garm. Garm stores the policy and policy key in memory for the duration of the current execution. Garm relies on operating system or hardware support of memory curtaining to ensure that attackers cannot inspect Garm's memory contents to obtain these keys. Garm is then responsible for enforcing the policy on the application's execution.

When an application makes a write system call, Garm intercepts the system call. It checks to see if the data access policies apply to any of the data to be written. If so, it encrypts the data at the byte granularity using the policy keys for the policies that apply to the byte. Then Garm performs the system call to write the data to the application file. Garm then updates the corresponding provenance shadow file with the provenance values for the data that was written.

## 2.1 Intended Usage

Defending from attacks mounted by technically-sophisticated insiders is known to be extremely difficult. However, in many cases data is compromised by accidental disclosure. The primary goal of Garm is to prevent accidental disclosure and make clear that any disclosures were intentional. In particular, the goal of Garm is to protect against accidental releases that occur from common events including: (1) a user copies data to external media and later loses the device, (2) a user accidentally prints confidential information, (3) a user accidentally pastes confidential informa-

tion into a document that is later publicly released, (4) a user accidentally emails confidential information to the wrong address, or (5) a computer is lost or stolen.

We wish to protect against accidental disclosures while allowing authorized users to freely exchange and use protected data. In particular, we want to allow authorized users to freely use the data for legitimate business including: (1) seamlessly sharing confidential data between authorized parties through email, external media, and network stores and (2) incorporating confidential data into documents. In many cases, whether a given action is allowed depends on future events. For example, we want to allow authorized users to store confidential data on a USB stick for legitimate purposes, but prevent later unauthorized uses of the data on that USB stick.

In the real world, programs process data with a wide-range of confidentiality requirements. For example, an email folder may store emails from a wide range of people each with different confidentiality requirements including completely public data. It is therefore important to support different usage policies for data within the same file and even the same program execution.

A secondary benefit of Garm is that it provides some protection from unsophisticated insiders in office environments. In particular, we assume that such insiders lack the resources to mount hardware attacks against the TPM and that large scale analog attacks such as taking repeated photos of the screen would be noticed.

Garm makes the following assumptions:

(1) Garm does not protect against buffer overruns and other attacks that compromise the integrity of trusted components.

(2) We assume data producers will use a white list of applications that the policy maker allows to use data.

(3) We assume that policy writers control their own policy servers and that they secure them against attackers.

(4) Garm does not protect against implicit leaks, timing, resource usage, RF emissions, or other side channels that may leak protected data. We expect that the use of white lists will partially address both hostile programs that are designed to explicitly use side channels to leak data and legitimate programs that leak data through implicit dependences.

(5) Garm is targeted largely towards legacy applications. Many legacy applications can incur large overheads and still be usable. We expect that new applications can be developed with native support for policy protection and tracking provenance.

(6) We assume the presence of trusted hardware support and a trusted operating system. We describe in more detail in Section 2.2.

(7) We assume that attackers do not have the resources to compromise trusted hardware.

### 2.2 Trusted Computing

Garm assumes that both the underlying operating system and hardware will provide trusted computing support. Future versions of Microsoft Windows are expected to include support for trusted computing in the form of the Next-Generation Secure

Computing Base system architecture [Peinado et al. 2004; England et al. 2003]. Trusted computing support consists of a trusted platform hardware module along with BIOS and operating system support.

Many proposed trusted computing systems support remote attestation for DRM. Remote attestation enables an application to prove both its identity and that it has not been tampered with to remote systems. Specifically, Garm needs to (1) prove to the policy servers that the underlying BIOS, boot loader, OS, hardware, and Garm installation is trusted, (2) provide a secure communications path from the server to the Garm installation, and (3) guarantee that confidential information will be destroyed or made inaccessible before the system performs any operations that may make it untrusted. This exact same problem is faced by any application (e.g. media players) that wishes to use remote attestation for DRM purposes.

Any number of approaches can provide these guarantees. For completeness, we describe one approach.

(1) The system begins with a standard trusted boot of the operating system using the TPM chip as follows. A secured loader hashes the BIOS image, extends the PCR register with this hash value, and then executes the BIOS. The BIOS reads the boot loader, hashes the boot loader image, extends the PCR register with this hash value, and then executes the boot loader. Finally, the bootloader reads the kernel, hashes the kernel image, extends the PCR register with this hash value, and then executes the kernel.
    Whenever the kernel loads a new driver, it first hashes it, extends the PCR with the hash value, and then runs the driver. We assume that all trusted drivers are signed, we explain the reasons later. Note that the kernel maintains a list of hash values that are used to extend the PCR.
(2) The operating system computes a hash of the Garm binary, extends the PCR value with this measurement, and then executes Garm.
(3) Next Garm loads. Garm then generates a public/private key pair for this session. It then calls the operating system to extend the PCR with a hash of the public key. The operating system's PCR extension mechanism must record that Garm initiated the extension.
(4) Garm contacts the policy server and sends the policy.
(5) The policy server responds by sending a randomly generated nonce as a remote attestation challenge.
(6) The OS loads a private key into the TPM. The TPM chip concatenates the nonce and the PCR register and signs both with its private key. The client computer then sends to the policy server: the signed nonce/PCR register, the public AIK key, the AIK certificate, the OS's list of hash values, and GARM's public key.
(7) The policy key server next verifies the following:
    (a) That TPM chip's public AIK key is valid.
    (b) That the signature for the PCR value is valid.
    (c) That the hash signatures correspond to PCR value.
    (d) That each hash signature represents a trusted component.

(e) That the hash of the public key from Garm matches the hash value used for the PCR.

(f) That the policy is satisfied by the GARM installation.

If all of the checks succeed, the policy server encrypts the policy key with the public key from the Garm installation and sends the policy key to Garm.

(8) Before either Garm or the operating system starts any operation that may compromise the system's trustworthiness, it must first clear all confidential information including Garm's private key. This requirement is why we require trusted drivers to be signed — it allows the OS to dynamically load drivers while the computer has been trusted with confidential information.

Clearing confidential information involves zeroing Garm's memory and extending the PCR register with a special value that indicates that the system is no longer trustworthy.

Note that once Garm is loaded, the kernel must record any events that may compromise Garm's trustworthiness. In particular, if a user runs a debugger on the Garm image, the kernel must log that event into the PCR value. Moreover, once Garm has generated a public/private key pair, before the OS performs any operation that may compromise its trustworthiness it must first zero all of Garm's memory. The intuition is that Garm must only possess the private key while it remains trusted.

Trusted computing also provides hardware support for sealing data for a specific application. Sealing allows the application to secure data that can only be accessed when the application proves its identity. Garm could use sealing to secure the private keys that it uses to sign the provenance shadow files to secure them against tampering. Trusted computing also supports curtaining an application's memory to prevent debuggers from obtaining the encryption keys from an application memory. Garm would use this feature to protect its private keys.

### 2.3 Protecting Data

When an application writes policy-protected data to a file, Garm must ensure that applications outside of the Garm framework cannot access the protected data. We next describe how Garm uses encryption to ensure that policy-protected data cannot be accessed outside of the policy enforcement framework.

2.3.1 *Cipher Choice.* Making encryption transparent to the application and enforcing access policies at a fine granularity places two constraints on our choice of ciphers. In particular, the cipher needs to support fast random access to large files and the ability to choose which bytes to encrypt at the byte granularity.

Garm uses Dan Bernstein's Salsa20 stream cipher as it satisfies both requirements [Bernstein 2008]. Salsa20 was one of the stream ciphers selected by the ECRYPT Stream Cipher Project [ecrypt 2008]. Salsa20 operates on 512-bit blocks. It takes as input a 256-bit key, a 64-bit nonce (unique message identifier), and a 64-bit block identifier. It generates as output a 512-bit cryptographically strong pseudo-random bit string which is xor'ed with the plaintext to encrypt the text or xor'ed with the ciphertext to decrypt the data. Salsa is notable in that it supports decrypting or encrypting a block in a file without requiring processing the previous blocks. Garm associates a nonce with each file and policy pair.

#### 2.3.2 *Encryption/Decryption.*

Garm associates a policy key with each data access policy. To write a byte, Garm lists all the policies that apply to that byte. It then looks up the nonces for each policy. Garm then generates the byte in the pseudo-random sequence that corresponds to the offset that the byte is written to. Garm then xors the byte at the given location in the keystreams for each policy with the byte to be encrypted. Finally, Garm writes the encrypted byte out to the file. Garm optimizes for the common case that adjacent bytes are protected by the same policies. Decrypting a byte uses the same algorithm.

One potential problem is that an application can write different bytes that are protected by the same policy to the same file at the same location. If the attacker observes both ciphertexts, the attacker has knowledge of $message_1 \oplus keystream$ and $message_2 \oplus keystream$ where $\oplus$ is the xor operation. If the attacker xors both ciphertexts, the attacker obtains $message_1 \oplus message_2$. If either message is known, the attacker can obtain the other message. Moreover, sophisticated analysis can often exploit redundancy in the messages to obtain both plaintexts. It is therefore imperative that a given key and nonce is only used at most once to write to a given file location. To address this weakness, Garm would conservatively monitor which locations a given nonce has been used to encrypt. If a location is ever repeated, Garm would generate a new nonce for the given policy and file. Garm can either then re-encrypt the data that uses the current nonce, or it can simply assign a special provenance value for the secondary nonce.

The algorithm as stated xors the keystreams. An alternative strategy is to xor the keys. This strategy is more efficient with multiple keys, but has the downside that changing a policy's nonce requires knowledge of all other policy keys that encrypt the same data. If these keys are not known, Garm could simply assign a special provenance value for the new nonce. Garm maintains a nonce shadow file that contains the nonces for each policy that applies to data in the primary file.

### 2.4 Authentication of Provenance Data

In some usage scenarios, it can be desirable to detect tampering with the provenance records or changes in the underlying files that are not reflected in the provenance data. This can be done by using cryptographic hash functions to compute a hash for the data file, including this hash in the corresponding shadow provenance file, computing a hash of the provenance file data, and then using a private key to sign the hashes. Garm would secure its private key by using the sealing functionality of the trusted platform module. Other instances of Garm could verify the signatures by looking up the public keys for the Garm installation that created the file, then verifying the signature with the public key.

## 3. PROVENANCE ANALYSIS

Garm uses a staged provenance analysis that uses a static analysis to generate an optimized dynamic analysis. It performs the instrumentation process on the binaries when the application is executed. We implemented Garm as an extension to the Valgrind binary instrumentation framework.

Figure 3 presents example code. Figure 4 presents how traditional binary rewriters would instrument this code to compute provenance or taint. The traditional approach is to generate, for each instruction, corresponding instrumentation that computes the effect of the original instruction on the data's provenance or taint. For

```
1 r1=a+b;
2 r2=a*b;
3 d=r1/r2;
4 e=a-b;
5 f=a+c;
6 f=f+b;
```

Fig. 3. Example Code

```
1  r1=a+b;
2  prov_r1=merge(prov_a, prov_b);
3  r2=a*b;
4  prov_r2=merge(prov_a, prov_b);
5  d=r1/r2;
6  prov_d=merge(prov_r1, prov_r2);
7  e=a-b;
8  prov_e=merge(prov_a, prov_b);
9  f=a+c;
10 prov_f=merge(prov_a, prov_c);
11 f=f+b;
12 prov_f=merge(prov_f, prov_b);
```

Fig. 4. Standard Instrumentation

```
1 r1=a+b;
2 r2=a*b;
3 d=r1/r2;
4 prov_d=merge(prov_a, prov_b);
5 e=a-b;
6 prov_e=prov_d;
7 f=a+c;
8 f=f+b;
9 prov_f=merge(prov_d, prov_c);
```

Fig. 5. Optimized Instrumentation

the example, the traditional approach would generate the provenance merge operations in lines 2, 4, 6, and 8 of Figure 4. We next describe how Garm's optimizations eliminate many of these instrumentation operations.

We present a technique that statically reasons about abstract provenances to generate optimized instrumentation. For the example code in Figure 3, our technique would determine that because the provenance of `r1` is the merge of the provenance of `a` and `b` and the provenance of `r2` is the merge of the provenance of `a` and `b` that the provenance of `d` is also the merge of the provenance of `a` and `b`. Our technique then generates the instrumentation code `prov_d=merge(prov_a, prov_b)` to compute the provenance of `d` and `prov_f=merge(prov_d,prov_c)`. Our static analysis determines that the provenance of `e` is the merge of the provenance of `a` and `b`. Our technique then determines that the provenance of `e` is equal to the provenance of `d`, and therefore it can simply reuse the provenance value for `d`. Our static analysis determines that the provenance of f is the merge of the provenance of `a`, `b`, and `c`. Our technique then determines that the provenance is also equal to the merge of the provenance of `d` and `c` and eliminates a provenance merge operation. If there is not an exact match, Garm searches for the largest computed provenance that is a subset of the current provenance and uses that provenance as a starting point to compute the new provenance. We next present the provenance analysis in more

detail.

### 3.1 State

Garm maintains shadow state of an application's state. The shadow stores a provenance value that describes a byte's current provenance. Each provenance value is an index into a provenance table of provenance descriptors. A provenance descriptor contains: a list of original source descriptors and a list of data policies. We next describe how Garm shadows the key components of an application's state:

- **Shadow Memory:** Garm contains a two-level shadow table: the first level is indexed by the high 16-bits of a memory address and the second level is indexed by the low 16-bits. Each entry gives the current provenance of the corresponding byte. There is a distinguished second level that is used for all unused memory blocks. When the application writes to an address, the corresponding second level table is allocated if necessary.
- **Registers:** Garm maintains the register values in a special memory region that serves as a register file. There is a shadow register file that contains the provenance of the current register values. Because several x86 registers can be accessed at 8-bit, 16-bit, and 32-bit granularities, Garm conceptually traces a register's provenance at the byte granularity. However, Garm optimizes for registers that can only be accessed as a full register by utilizing only a single provenance for such registers. Due to the details of how x86 registers can be accessed, 3 provenance shadows suffice for any register. For instance, even though the register %eax can be accessed in four different ways (%al, %ah, %ax, and %eax), there is a partition of the bits into 3 groups such that the bits accessed on any read is a union from these groups.
- **Temporaries:** Valgrind's intermediate representation introduces several temporary variables. Garm conceptually traces the provenance of temporary variables using a single shadow provenance regardless of the temporary variable's actual length. Garm's runtime instrumentation does not explicitly shadow these temporaries. The temporaries are analyzed by the static analysis, and the dynamic instrumentation code simply updates the shadow register file and memory.
- **Files:** Garm maintains shadow files for all files that a program accesses. The shadow files store the provenances of each byte in the original file.

### 3.2 Optimized Instrumentation

Our approach uses a data flow analysis to compute the abstract provenance of each temporary at each program point. Garm uses the analysis to generate an optimized dynamic instrumentation by eliminating redundant provenance computations. Redundant provenance computations occur when the same value is repeatedly combined with an existing value. We represent an abstract provenance as a tuple consisting of: (1) a set of load instructions and other instructions that contribute to the provenance, (2) a set of register locations that contribute to the provenance, (3) a subset of the set of shadow temporary variables that contribute to the provenance, and (4) the provenance contribution from the instructions. Note that the exact provenance contribution from the code for an instruction is known statically. Garm obtains instruction provenance values from the shadow provenance

file for the executable if it is available. Such a provenance shadow file is generated if Garm monitored the compilation of the application or someone applied a policy to the application's binary.

We model the registers' provenance using the partial function *registers* that maps a register to either an abstract provenance or a temporary that holds the abstract provenance. We record the provenance of the shadow temporaries using the partial function *shadowvalues*. An undefined provenance for a register indicates that the register's provenance is statically unknown. We model the temporary variable's provenance using the partial function *temporaries*. The provenance state at a program point is characterized by the tuple consisting of the provenance of the register file *registers*, the provenance of the temporary variables *temporaries*, and the provenance of the provenance shadow temporaries *shadowvalues*.

The analysis begins with an initial tuple with empty partial functions. It then executes the instructions in the superblock[1] on the abstract provenance values. For example, a move instruction sets the destination temporary variable's provenance to the combination of the source temporary variable's provenance and the move instruction's provenance.

As Valgrind's superblock-based intermediate representation does not allow placing instructions after a conditional exit, Garm must ensure that the provenances in the shadow register file are current at every conditional exit instruction. The instrumented code generates the appropriate provenance values in temporaries and writes them to the register file. Therefore, the conditional exit instruction replaces the abstract provenances in the abstracted shadow register file with temporaries that hold these generated provenances.

As the static analysis is computed on linear sequences of instructions, there is no need for fixed-point computations or merge operations. The analysis simply processes the instructions in order in a single pass.

### 3.3 Dynamic Analysis

We next describe how Garm uses the static analysis results to generate dynamic instrumentation. The first step is to use the analysis results to compute the set of operations that must be instrumented. We use a workset-based instrumentation selection algorithm. This algorithm begins with the instructions in a superblock that can affect the provenance of the state that persists across superblocks. These instructions include stores to the memory and the values of registers at the exits. The algorithm then works backwards from these instructions to compute the set of load instructions and other instructions that must be instrumented. The result of the analysis is a set *toinstrument* of provenance source instructions that must be instrumented.

3.3.1 *Instrumenting Sources.* We next describe how the instrumentation algorithm processes instructions that may appear as a source for calculating a provenance. When the instrumentation algorithm visits a load instruction to be instrumented, it generates a shadow load instruction that loads the corresponding provenance from the shadow memory. When the instrumentation algorithm processes a register load instruction, it checks to see if the register offset appears in

[1]A superblock in Valgrind is a linear sequence of instructions with a single entrance and many possible exits.

the *toinstrument* set and has not already been loaded. If the register load meets both criteria, the instrumentation algorithm generates a shadow register load that loads the register's shadow provenance value into a temporary variable.

3.3.2 *Computing Provenances.* We next describe how Garm uses the results from the static analysis to generate optimized code that computes dynamic provenances. The algorithm begins with the provenance contribution from the program's instructions. It represents this provenance with a 32-bit provenance value. The algorithm then generates instructions to merge the provenance contributions from the set of load and other instructions. These provenances will already be stored in temporaries as the corresponding instructions will have already been instrumented. The algorithm next generates instructions to merge the provenance contributions from the set of shadow register values. These provenances will also already be stored in temporaries as the corresponding register load operations will have already been instrumented. Finally, the algorithm generates instrumentation code that merges the provenance contributions from the temporaries in the shadow temporary set.

Garm uses the information in the shadow temporary set to further optimize the instrumentation code. The shadow temporary set stored in the partial function *shadowvalues* records the set of shadow temporaries for which Garm has already computed a provenance and their corresponding abstract provenance. Before Garm generates a provenance it first checks to see if it has already generated instrumentation for an abstract provenance that is a subset of or equal to the provenance to be generated. It then uses the results from the existing provenance computation as a starting point for the new computation. This allows Garm to elide all or part of the instrumentation to compute the provenance.

3.3.3 *Storing Provenances.* Finally, Garm instruments instructions that write values to memory or write values to registers that may be visible outside of the superblock. Garm instruments memory stores by using the algorithm from Section 3.3.2 to generate code that computes the provenance for the source and address temporaries. Then it generates code that stores this provenance into the shadow memory.

### 3.4 Intercepting Data Accesses

Garm intercepts systems calls that open new files, read from a file descriptor, or write to a file descriptor. It rewrites open system calls to record which file was opened and if possible to create a corresponding provenance file. Note that because Linux uses the file abstraction for accessing many hardware devices, this same mechanism will also create provenance entries for devices that the program accesses.

Garm rewrites read system calls to also read the provenance information from the corresponding provenance file. If a policy applies to the data and it allows access, Garm will also decrypt the data. Garm rewrites write system calls to also write the provenance information to the corresponding provenance file. If a policy applies to the data and it requires encryption, Garm will also encrypt the data. Because Linux exposes many hardware devices to applications through the file interface, these same mechanisms track the provenance of data accessed through the console or other devices.

While the current implementation does not support tracking provenance across TCP/IP connections, it is straightforward to extend Garm to intercept TCP/IP

connect system calls, establish a second TCP/IP connection to the another Garm instance, and communicate provenance data through this connection. This mechanism would allow Garm to protect data sent between arbitrary programs and track its provenance.

3.4.1 *Provenance Representation.* We next describe Garm's representation of data provenances.

3.4.1.1 *Base Provenances.* Garm uses *base provenances* to trace the sources of incoming data. The first time that any application under Garm accesses data from a given file, Garm records a base provenance that contains the file name, the current execution number, and a type that denotes that this is the first time the Garm system has seen this data. If an application under Garm accesses data that was modified by a previous application under Garm, Garm records 1) the execution identifier for that previous execution, 2) the file name, 3) a reference to the 32-bit provenance identifier from the previous execution, and 4) a list of references to all access policies that apply to the data. Garm represents base provenances as a 32-bit index into the base provenance table. Each entry in the base provenance table gives the complete description of the provenance. Garm maintains the invariant that two identical base provenances must have the same index.

3.4.1.2 *Composite Provenances.* The execution of an application performs operations that combine data from multiple sources to produce derived values. These operations produce data whose provenance is derived from all of the relevant data sources' provenances. Garm uses *composite provenances* to trace the set of source base provenances that contribute to the provenance of the application's state. A composite provenance consists of a list of base provenances. Garm also records the list of references to the policies that apply to the composite provenance for efficiency reasons. Garm represents composite provenances as a 32-bit index into the composite provenance table. Each entry in the composite provenance table lists the component base provenances and any applicable data policies. Garm maintains the invariant that two identical composite provenances must have the same index.

Conceptually, an individual byte's provenance can be viewed as a directed acyclic graph. The graph is acyclic as each dependence must respect causality — the inputs to an execution must have been written before the execution read them. Each composite provenance can reference several base provenances. Each base provenance can in turn reference a composite provenance from a previous execution. Garm contains a provenance query tool that allows users to explore the provenance tree of an application's output files.

The current implementation of Garm does not support multiple Garm invocations simultaneously accessing the same file. Future work can eliminate this limitation.

3.4.1.3 *Merging Provenances.* The primary operation that the instrumented code performs is merging multiple source composite provenances into a single output composite provenance. We next describe the basic algorithm for merging two provenances:

1. **Identity Check:** The merge procedure first checks if the input composite provenance indices are the same and if so simply returns that composite provenance index.

2. **Merge Cache Lookup:** The merge procedure next performs a table lookup on the two input composite provenance indices to see if the merge result provenance has been cached. If the result is stored in the cache, the algorithm simply returns the cached composite provenance index.
3. **Base Merge Procedure:** Otherwise, the base merge procedure begins by looking up the two input indices in the composite provenance table. As Garm stores composite provenances as sorted lists of references to base provenances and policies, it merges the two composite provenances. The algorithm then looks up the merged composite provenance in the composite provenance hash table. If the composite provenance is not found, the algorithm adds the new composite provenance to the table and generates a new composite provenance index. Otherwise, it uses the composite provenance index from the hash table lookup. Finally, it caches the results of the merge operation in the merge cache — it stores the input composite provenance indices and the output composite provenance index.

## 4. LIMITATIONS

We next discuss limitations of Garm and approaches to address them. These limitations may not be applicable to all Garm usage scenarios or deployments. For example, many limitations are not a concern for scenarios in which Garm is simply used to prevent an accidental release of medical records through a lost USB memory key.

### 4.1 Implicit Flows

Garm only traces explicit flows of information. Information about the contents of data can also leak through implicit channels including control flow or timing channels.

Garm does not trace implicit flows due to the extra dynamic overhead and the imprecision that can be introduced. Such implicit flows can be used to attack the policy enforcement mechanism of Garm, an attacker can write a program that uses implicit flows to copy data without copying the policy.

We intend that users mitigate this class of attacks through data access policies that restrict the applications that can access the data. We expect that policies will use one of two basic strategies: white listing applications or signing applications. The first approach uses a white list of secure hashes of applications that are known to not be malicious. When Garm executes a binary, it would compute the binary's hash and then check whether the binary's hash was white listed. An organization might use this approach to allow data to be used with a standard suite of applications. Self-signed executables could be used to allow updates to white-listed applications [Wurster and van Oorschot 2009].

A second approach utilizes binary signing. The idea is that a trusted authority would provide certificates to trusted software development companies who would then sign their binaries with these certificates. If a certificate was compromised, the Garm installation would be updated with a list of compromised certificates to invalidate. Individual policies could require that Garm certify that it has obtained the latest revocations from the certificate revocation server and could list which certificates are trusted.

### 4.2 Provenance Data

Provenance data can itself leak protected information. Consider the example of a protected vector image that is rendered on a white bitmap. The rendering of the protected vector image could easily only touch the image locations that should be colored black. The provenance shadow information would then reflect what locations had been colored black and provide information about the original image.

A countermeasure to this class of attack is straightforward — policies can limit the amount of information that the provenance data encodes. The idea is to compute how much information is represented by the provenance data and then coarsen the provenance information to meet policy thresholds.

### 4.3 Compromise of Trusted Computing Infrastructure

Historically, hackers have discovered exploits for most trusted computing systems. An important design consideration is limiting the damage caused by an exploit. If a Garm client is compromised, the hacker can obtain policy keys for all data that the Garm client has access to. Note that as soon as the compromise is discovered, the policy servers can refuse to send the compromised client any more policy keys. The policy servers will also have a detailed log that captures the scope of compromise. Moreover, before sending keys the policy server verifies that the Garm installation has permission to access the policy-protected data in some fashion. Therefore, the effect of an exploit is that attackers can only circumvent policies for data for which they already have some type of access.

### 4.4 Interpreters

The basic techniques of Garm work on interpreters. However, an interpreter can takes as input a program that creates an implicit channel to leak protected data. Therefore, white listing an interpreter can compromise protected data. One approach to address this issue is to have a white list specify both the interpreter and the input file that is trusted.

Many applications like Microsoft Word contain embedded interpreters. One approach to addressing the interpreter issue for these types of applications is to have a white list specify both an executable and a provenance file for that executable. The provenance file would then taint the machine code for the interpreter component of the application. The effect is that any data outputted by the interpreter would automatically be tainted and therefore hostile script files could not run in a trusted application to declassify policy-protected data.

## 5. EXPERIENCE

We next discuss our experience using Garm to trace provenances and enforce data access policies with several applications. We have developed a prototype implementation of Garm. Our implementation consists of approximately 8,500 lines of C code. Our prototype implements a limited set of policies: access once, unencrypted output to the audio device, unencrypted output to the screen, unencrypted output to any source, and encrypted only output. It is straightforward to extend Garm with new classes of policies through minor changes to the source code. Garm provides the basic mechanism that when combined with a policy language would allow end users to specify a rich set of customized data usage policies.

Our prototype implements the staged provenance analysis along with the stream

cipher. While it is conceptually straightforward, the prototype does not interface with the operating system to support remote attestation or sealing keys.

We ran Garm on a workstation with a 2.4 GHz Core 2 Duo processor, 1 GB of RAM, and Debian Linux running kernel version 2.6.30.

### 5.1 Data Provenance

We first discuss our experience using Garm to trace provenances across the executions of several command line utilities included with the Debian Linux distribution. In particular, we used Garm to trace the provenance of data across executions of gzip, tar, nano, vi, Alpine, and sort. After each execution, we used Garm's provenance viewing utility to manually examine the provenance of the output files.

In our first experiment, we used Garm to protect emails in the Alpine email client, an open source version of the classic Pine client. Alpine stores all emails in an email folder in the same file. Therefore, precisely tracking the policies of individual emails in Alpine requires tracking provenances at a sub-file level granularity. We included text from a policy-protected file into an email in Alpine. We then sent the email and confirmed that Garm's policy-protection mechanism resulted in the policy-protected text being encrypted before being sent to the server. We then modified the policy-protected email to create a derived email. We next added a non-protected email to the same folder. We then reloaded Alpine and confirmed that we could still view the policy-protected emails in the folders. We also confirmed that the policy-protected parts of the emails were properly encrypted on the disk and that the non-protected email was visible as plaintext. The current version of Garm does not contain support for communicating the provenance of data sent across TCP/IP and therefore the policy-protected text can not currently be read by the recipient. However, future versions could communicate this data using a secondary connection and would allow applications to seamlessly share policy-protected data.

In our second experiment, we used Garm with vi and nano, two interactive text editors. We edited a file during several sessions of both the vi and nano text editors. Afterward, we viewed the provenances of the characters in the text file using Garm's provenance viewing utility and verified their correctness. Garm was able to successfully trace the provenance of each byte of our text file across the editing sessions. In particular, Garm correctly identified, for each byte, the session that byte was entered and listed each subsequent program execution that manipulated the text.

We used tar to archive several files. We then decompressed the archive. We manually verified the provenances of the output files. Garm was able to precisely trace provenances of data across both tarring the data into a tar archive and untarring the data into files.

We used gzip to compress and then decompress the same text file. We then examined the provenance of the output. We observed that the provenances were conservative. However, we did observe some imprecision in the provenances introduced in the process — the provenance of a few bytes included extra editing sessions. This imprecision is an artifact of compression — the compression algorithm may extract redundancy across editing sessions and in the process mix the provenances.

Finally, we used sort to sort a text file developed over several sessions. We

observed mixing of provenances, but in this case the location of a text line depends on the other lines and the mixing of provenances correctly reflects this dependence.

### 5.2 Policy Enforcement

Our Garm prototype supports a limited set of policies including access once, allow viewing text on the terminal, and allow playing through the audio device. We used Garm's policy tool to protect text files, source code, and MP3 files.

We used Garm with mpg123, an MP3 player, to play a protected MP3 file. We created data policies that used many combinations of the basic access policies. For example, we created MP3 files that could be played once, MP3 files that could be played but whose song name could not be viewed, and MP3 files whose song names could be viewed but that could not be played through the audio device. We found that Garm was able to successfully enforce the policies and that Garm had sufficient performance to run mpg123 in real-time.

We protected text files with policies that allowed viewing them exactly once. We then viewed the file with the nano editor and added new text. We then attempted to view the file a second time and observed that we could view the new text but not the policy-protected text.

We protected C source files with a policy that does not allow viewing the code on the screen. We then used gcc to compile the policy-protected C source code into a binary and verified that the binary was similarly protected. Finally, we instructed the policy server to release the specific policy from the binary and then verified that the binary executed correctly.

### 5.3 Overheads

We next evaluate both the execution time and space overheads of Garm.

5.3.1 *Execution Time.* We measured the overhead of Garm on several representative benchmarks applications. We include numbers for Garm for the CSE version with the CSE optimization and for the base version with the CSE optimization disabled. We include for comparison overheads for the base Valgrind implementation (memcheck).

Figure 6 presents the measured slowdown ($\frac{t_{\texttt{Garm}}}{t_{\texttt{Normal}}}$) for Garm on each benchmark. In the first benchmark, we measured the time take to decompress a gzipped file. We measured the overhead of Garm to be a factor of 5.64. In the second benchmark, we measured the time taken to extract files from a tar archive. We measured the overhead of Garm to be a factor of 5.34. The overhead of this benchmark is primarily due to the extra shadow information that Garm must record. The third benchmark uses mpg123 to decode an MP3 file. We measured the overhead of Garm to be a factor of 13.14.

| Benchmark | CSE | Base | Memcheck |
|---|---|---|---|
| gzip decompression | 5.6× | 6.3× | 2.4× |
| tar extracting | 5.3× | 5.3× | 1.3× |
| mpg123 | 11.4× | 13.2× | 6.8× |

Fig. 6. Slowdowns

To qualitatively evaluate the performance impact of Garm, we monitored sev-

eral interactive applications including bash, xdvi, pico, nano, ssh, scp, and other command line utilities with Garm. We found that slowdown was barely noticeable with these benchmarks and that the overhead of Garm did not significantly affect interactive performance. We expect that Garm provides capabilities that can benefit many users and that the users of many applications can tolerate the current overheads of Garm.

The design of Valgrind introduces a number of significant overheads to Garm. We discuss several of the more significant overheads below.

—**Lack of Conditional Support in Valgrind IR:**
In Valgrind, methods calls are the only way to add instrumentation code with conditional branches. The common case for merging two provenances is to compare the provenances with the cmp instruction, find them to be the same, and therefore simply copy the provenance. If Valgrind provided a more expressive intermediate representation, the common case for the provenance merge operation could be implemented with two instructions: a compare instruction and a conditional branch instruction. However, Valgrind forces us to use a method call that makes provenance tracking much more expensive.

—**Short Blocks:**
Our instrumentation optimizations are hindered by the relatively short sequences of instructions that Valgrind presents to our optimization pass. These short sequences limit the gains our optimizations can realize — these short sequences typically store the results of all their computations into registers. This makes it impossible for our optimizations to eliminate many provenance computations. Moreover, it also means that the instrumentation algorithm must assume the worst case for registers that are used across superblock boundaries. Even though the code is likely to only access the register at the word granularity, the analysis must assume that out of context code may access the registers at the byte granularity.
An intermediate representation that exposed longer code sequences to our analysis would yield greatly improve code. In particular, the analysis would avoid most of the provenance merge operations that are performed because of the worse case assumptions about register usage.

—**Superblock Form:** Valgrind exposes code to transformation phases as super blocks that consist of straight line code with a number of conditional exit points. These superblocks do not allow for code that is conditionally executed if an exit is taken. Therefore our analysis must ensure that the provenance shadows are correct at the entrance to every conditional branch instead of conditionally updating them if the branch is actually taken.

Inspection of the instrumented code reveals that many of the provenance merge operations serve to merge the provenance of bytes that comprise a word of memory. We expect that Garm's overhead can be significantly reduced by tracing provenance at a coarser granularity. For example, tracing provenances at the word granularity would eliminate a large number of reads, writes, and provenance merge operations. It would also decrease Garm's space overhead to only a one hundred percent increase in memory usage. This would significantly reduce both Garm's space and computational overheads.

5.3.2 *Space Overheads.* The current implementation of Garm imposes a four hundred percent increase in the amount of memory space used by a program. While this is a significant overhead, in practice for programs that originally consumed less than a gigabyte of memory it represents a small investment in memory with current memory prices.

While Garm maintains shadow files that are four times the size of the original files, they contain significant redundancy. We found that gzip can reduce the size of the shadow files by a factor of 1,000 (for a total overhead after compression of 0.4%). We expect that off-the-shelf compression online algorithms will make both the time and space overheads of the shadow files insignificant.

Note that for performance critical applications, it is straightforward to use operating system level mechanisms to provide very coarse granularity, but extremely lightweight provenance tracing and policy enforcement mechanisms. Such mechanisms can be designed to use Garm's shadow provenance file format, and therefore allow performance critical applications to access Garm-protected data.

## 6. RELATED WORK

Understanding how information flows through applications is an active area of research. Most of the research has focused on analyzing a single application to understand whether it leaks secret information or potentially uses untrusted information in an unsafe manner.

Static information flow attempts to show that applications do not leak confidential information [Denning 1976]. Static information flow analysis requires that the developer know *a priori* what information is confidential and what information sinks are public.

### 6.1 Taint Analysis

Taint analysis traces whether data in an application is tainted. Depending on the application, tainting can be used to represent that information is secret and must not be leaked or that information is from an untrusted source and must be carefully checked before it is used.

Researchers have developed a large number of taint analyses. Taintcheck uses binary instrumentation to trace whether bytes in an application are tainted [Newsome and Song 2005]. Conceptually, it maintains a shadow bit for each byte in the application that indicates whether that byte is tainted. It uses a set of security policies that describe which sources are tainted and restrict how tainted values may be used. DYTAN [Clause et al. 2007] and Flayer [Drewry and Ormandy 2007] also use binary instrumentation to implement tainting. Haldar et al. have implemented dynamic tainting for Java to ensure safety of web-based applications [Haldar et al. 2005]. McCamant and Ernst have developed a dynamic analysis technique that uses flow networks to quantify an upper bound on the number of bits that leak [McCamant and Ernst 2008]. The RIFLE [Vachharajani et al. 2004] project explores architectural extensions to efficiently support dynamic analysis for information flows.

The Raksha project provides hardware support for simultaneously tracing tainted bits for a handful of security policies [Dalton et al. 2007]. They use this functionality to discover attacks on an application. Chandra and Franz have implemented a dynamic analysis for Java that implements a fine-grained labeling scheme [Chandra

and Franz 2007] that provides support for application-level policies. Their approach supports up to 32 hierarchical security levels.

Trishul [Nair et al. 2008] is a virtual machine with support for information flow-based policy enforcement. Trishul traces tainting information while Garm traces provenance. Trishul enforces policies on how specific applications access system files while Garm allows a user to specify policies on how data can be used by arbitrary binaries.

Garm has significant differences with taint analysis. In particular, taint analyses are very coarse-grained. They typically allocate only 1 bit per value. Therefore tainting makes the implicit assumption that the user knows *a priori*, which sources of values should be tainted. Because Garm assigns unique provenance records whenever it loads data from a different source, it records complete provenance information and does not require that the user identify tainted sources ahead of time.

Most current taint frameworks do not trace taint information across application boundaries. Because of this limitation, they are forced to decide at the boundary of the tainting framework whether to completely trust the recipient with the unprotected data or to block transmission of the data [Hicks and McDaniel 2007]. This limitation has a profound impact — with such systems a single location cannot be used to store both protected and unprotected data.

The combination of encryption and provenance shadow files allows Garm to write policy-protected data to locations that can potentially be accessed by untrusted entities and guarantee that the policies will continue to be enforced when future executions of any applications access the data. We note that a few tainting frameworks can trace taint across application boundaries by performing taint analysis on a virtual machine [Chow et al. 2004; Yin et al. 2007] — this approach suffers from the same problem when applications write data to disk or send data across the network.

In general, taint frameworks support policies that specify how a specific application can use data from an outside source. Garm supports an entirely different class of policies — Garm's policies describe how data can be used by any application and not how a specific application can manipulate data. This enables Garm's policies to protect data across applications and machine boundaries and specify how arbitrary applications can use the policy-protected data.

An early workshop paper overviewed the approach [Demsky 2009]. This article provides the technical details, describes the integration of the approach with a TPM, describe the analysis and optimization approach, and more extensively evaluates the work.

### 6.2 Information Flow-Based Security

The HiStar operating system uses information flow to minimize the amount of code that must be trusted [Zeldovich et al. 2006]. It allows applications to create taint categories. All objects and threads in HiStar have a taint level for each category. For a given taint category, threads cannot read from objects with higher tainting values than the thread or write to objects with lower tainting values. Special wrapper programs are used to declassify the output of a program.

HiStar requires software to be rewritten to support information flow while Garm

can operate on legacy binaries. Execution threads must explicitly request to change their taint level before reading tainted data. The entire thread remains tainted — once a thread raises its taint level it cannot write even unclassified data to objects with lower taint levels. For applications in which this is undesirable, the developer must spawn a separate process to access the tainted data. For example, a word processor in HiStar would have to be architected to spawn a separate editing process for each document with different taint values.

Histar uses a much coarser-grained tainting mechanism for files that Garm. For example, a tar utility in HiStar would have to taint the tar file with the highest taint level of the individual files that comprise the tar file. When a file is later extracted from the tar file, it may no longer be readable by the user or process that conceptually owns it. Moreover, it would not be possible to write an email client that stores messages with different taint levels in the same file.

Finally, HiStar does not protect data on a machine or external storage device if the machine is lost or stolen. HiStar is primarily designed to protect a user from hostile programs or external attackers while Garm is primarily designed to protect data from accidental disclosure. Note that HiStar would not protect a user from accidentally sending confidential data to the wrong person.

DStar labels processes in a distributed system with communication permissions and the set of privileges the process can be used to omit restrictions on sending messages [Zeldovich et al. 2008]. The system uses the labels along with a set of policies to mitigate the consequences of untrusted code.

JiF [Sabelfeld and Myers 2003] is an extension to Java that adds support for information flow and policy enforcement. The basic idea is to write policies that partition output files as trusted or untrusted and then only allow the application to write tainted data to the trusted files. Garm's cross application provenance (and provenance-based encryption) allows applications to write policy-protected data to any file while still tracing the data's provenance and enforcing the data's access policy. This capability is key for enabling information-flow based security in the modern work environment as the widespread adoption of information-flow based security may ultimately depend on not burdening users with onerous restrictions on how they use data.

JPMail [Hicks et al. 2006] is a mail application developed in JiF that was manually coded to support sending policy-protected emails through untrusted servers. This approach requires the developers of a specific application to implement an entire policy framework and policy server. Garm basic approach extended to support TCP/IP could provide this type of policy support for arbitrary client-server applications without requiring any modification of the applications.

### 6.3 Data Provenance

Lin et al. introduces a dynamic provenance analysis based on reduced order binary decision diagrams [Lin et al. 2008]. Their analysis can compute, for each byte in an application's execution, the set of all input bytes that byte depends on. The analysis represents provenances as indices into a table of binary decision diagrams. The extra precision causes their analysis to incur heavy overheads. Their approach could potentially benefit from the static analysis in Garm.

Database researchers have studied the problem of tracing and maintaining prove-

nance in databases [Buneman et al. 2000; 2001]. Garm presents a technique that can trace provenance on arbitrary binaries at a level of abstraction that captures sufficient information to easily trace the source while simultaneously not incurring excessive overheads. Researchers have developed automated provenance gathering frameworks that operate at the file granularity [Muniswamy-Reddy et al. 2006] — for each outputted file they record the application that created the file, how it was invoked, and a list of all files it read. Hasan et al have developed library level tools to produce verifiable provenance records for files [Hasan et al. 2009]. Our approach is much more precise — Garm can determine which of multiple inputs (at the byte granularity) to an application contributed to a given output file.

We note that Garm could be modified to support the secure provenance chains describe by Hasan. This would require keeping all of the old provenance shadow files. Garm would then record a hash for each of the input files in the provenance record and then sign each of its output files. This would create a chain for the provenance records with the same properties as Hasan.

### 6.4 Policies

PinUP can enforce policies on how applications use files at the file granularity [Enck et al. 2008]. The basic approach adds kernel support to allow a machine's owner to specify the set of applications that can access files of a given type. For example, such a policy might allow Word but not other applications to access documents. This approach can restrict normal business processes including emailing documents and fails to differentiate between confidential and public documents of the same type.

Researchers have explored policy languages that restrict how data can be used. Pretschner et al. introduce a usage language that restricts how data consumers use data [Pretschner et al. 2008]. OSL [Hilty et al. 2007] and ODRL [Iannela 2002] both restrict how data can be used. Our work provides a framework that can enforce such policies on arbitrary binaries.

## 7. CONCLUSION

In current software systems, it is difficult to discover the history of how a file has reached its current state and it is difficult to control how files are used when sent to others. Garm uses a staged analysis that combines static and dynamic analysis to trace the provenance of data across applications. Garm provides a set of tools to query the provenance of data. This information can be useful for auditing purposes. For example, an organization might use the data to understand the scale of the consequences of a software error.

Garm can also use the provenance analysis to label data with access policies. Garm can then enforce these policies across application boundaries. These access policies might ensure that personal health records do not accidentally leave an insurance company's office computers while allowing the insurance company's employees to use the medical data with the software applications required to do business.

## REFERENCES

Bernstein, D. J. 2008. *New Stream Cipher Designs: The eSTREAM Finalists*. Springer, Chapter The Salsa20 Family of Stream Ciphers, 84–97.

Buneman, P., Khanna, S., and Tan, W.-C. 2000. Data provenance: Some basic issues. In

*Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*.

Buneman, P., Khanna, S., and Tan, W. C. 2001. Why and where: A characterization of data provenance. In *Proceedings of the 8th International Conference on Database Theory*.

Chandra, D. and Franz, M. 2007. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Twenty-Third Annual Computer Security Applications Conference*.

Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium*.

Clause, J., Li, W., and Orso, A. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*.

Dalton, M., Kannan, H., and Kozyrakis, C. 2007. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Intl. Symposium on Computer Architecture*.

Demsky, B. 2009. Garm: Cross application data provenance and policy enforcement. In *USENIX 2009 Workshop on Hot Topics in Security (HotSec)*.

Denning, D. E. 1976. A lattice model of secure information flow. *Communications of the ACM 19,* 5, 236–243.

Drewry, W. and Ormandy, T. 2007. Flayer: Exposing application internals. In *Proceedings of the First USENIX Workshop on Offensive Technologies*.

ecrypt 2008. The eSTREAM project. `http://www.ecrypt.eu.org/stream/`.

Enck, W., McDaniel, P., and Jaeger, T. 2008. Pinup: Pinning user files to known applications. In *Proceedings of the 24th Annual Computer Security Applications Conference*.

England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. 2003. A trusted open platform. *Computer 36,* 7, 55–62.

Haldar, V., Chandra, D., and Franz, M. 2005. Dynamic taint propagation for Java. In *21st Annual Computer Security Applications Conference*.

Hasan, R., Sion, R., and Winslett, M. 2009. The case of the fake picasso: Preventing history forgery with secure provenance. In *Proccedings of the 7th Conference on File and Storage Technologies*.

Hicks, B., Ahmadizadeh, K., and McDaniel, P. 2006. Understanding practical application development in security-typed languages. In *Proceedings of the 22st Annual Computer Security Applications Conference*.

Hicks, B. and McDaniel, P. 2007. Channels: Runtime system infrastructure for security-typed languages. In *Proceedings of the 23rd Annual Computer Security Applications Conference*.

Hilty, M., Pretschner, A., Basin, D., Schaefer, C., and Walter, T. 2007. A policy language for distributed usage control. In *Proceedings of the 12th European Symposium on Research in Computer Security*.

Iannela, R. 2002. Open digital rights language - version 1.1. `http://ordl.net/1.1/ODRL-11.pdf`.

Lin, Z., Zhang, X., and Xu, D. 2008. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *In Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

McCamant, S. and Ernst, M. D. 2008. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. Tucson, AZ, USA, 193–205.

Mitchell, C. 2005. *Trusted Computing*. The Institute of Electrical Engineers.

Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. 2006. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*.

Nair, S. K., Simpson, P. N. D., Crispo, B., and Tanenbaum, A. S. 2008. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science 197,* 1, 3–16.

Newsome, J. and Song, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commidity software. In *Proceedings of the Network and Distributed System Security Symposium*.

Peinado, M., Chen, Y., England, P., and Manferdelli, J. 2004. *NGSCB: A Trusted Open System*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 86–97.

Pretschner, A., Hilty, M., Basin, D., Schaefer, C., and Walter, T. 2008. Mechanisms for usage control. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*. ACM, 240–244.

Sabelfeld, A. and Myers, A. C. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication, special issue on Formal Methods for Security 21,* 1, 5–19.

Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., and August, D. I. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*.

Wurster, G. and van Oorschot, P. 2009. Self-signed executables: Restricting replacement of program binaries by malware. In *USENIX 2009 Workshop on Hot Topics in Security (HotSec)*.

Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*.

Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

Zeldovich, N., Boyd-Wickizer, S., and Mazières, D. 2008. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*.

Received Month Year; revised Month Year; accepted Month Year