Garm: Cross Application Data Provenance and Policy Enforcement

Brian Demsky
University of California, Irvine

Abstract

We present Garm, a new tool for tracing data provenance and enforcing data access policies with arbitrary binaries. Users can use Garm to attach access policies to data and Garm ensures that all accesses to the data (and derived data) across all applications and executions are consistent with the policy. Garm uses a staged analysis that combines a static analysis with a dynamic analysis to trace the provenance of an application's state and the policies that apply to this state. The implementation 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.

1. Introduction

Today's computing systems typically do not store information about the provenance (i.e. history) of files. In particular, users cannot query about the information sources or the chain of people and programs that contributed to a file's creation. The lack of information about files' provenance profoundly affects computing. For example, it can be difficult to detect malicious tampering of files in today's systems. It can also be difficult to verify the origin of digital works — for example, auditing whether an application's source code contains code misappropriated from other projects is generally challenging. Knowledge of data provenance can be useful even in the absence of malicious actions. For example, if one discovers a software bug that causes cryptographically weak keys to be generated, data provenance can help determine which keys are affected.

With current computing systems once information is released to others, there is no control over or knowledge about how that information is used. For example, providing personal information to any entity comes with 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. These 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 Garm, a tool that uses binary-rewriting to track the provenance of the data applications process and to enforce access policies on this data. Garm provides a framework that allows users to specify policies for their data and Garm ensures that all applications respect these policies. Unlike previous work, Garm's data access policies follow the protected data across execution, application, and machine boundaries.

Garm allows off-the-shelf binaries to process protected data while a dynamic analysis tracks both the provenance of an application's data and which policies apply to the data. The rewriting system enforces the policies, thereby allowing policy-protected data to be used with arbitrary applications while still enforcing the policies. The ability to enforce policies across a wide range of applications has potential benefits ranging from the security of personal data to more user-friendly and flexible digital rights management.

Previous work on information flow required access policies to specify which locations trusted data can be stored and then assumed all data from these sources is trusted. Garm supports a more flexible model — applications can write policy protected data to any location. Garm tracks the policies that apply to a program's output files through the use of provenance shadow files and uses fine-grained encryption to prevent accesses to the data that circumvent the policies.

1.1 Basic Approach

Garm uses binary rewriting to instrument binaries to trace 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 the guest application accesses new data, Garm creates a base provenance record to describe the source of the data. If Garm had previously monitored the program execution that produced the data, the base provenance will reference the provenance record from the previous execution. When the monitored program performs operations that depend on data with different

provenance, Garm labels the bytes produced by the operation with a composite provenance that lists the base provenance values that contributed to the current value. In this fashion, Garm computes the shadow composite provenance for all bytes that the application writes.

When an application writes data to a file, Garm generates a shadow file that describes the provenance of every byte in the file. Garm also stores a provenance record that describes the source for each base provenance and lists the base provenance records that contribute to each composite provenance.

Garm supports data access policies by allowing users to attach a data access policy to a provenance. 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 in 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 policy can depend on context (i.e. the date, how many times the data has been accessed, etc). Garm uses fine-grained encryption based on stream ciphers to enable 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 determine 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 track the provenance of an application's state. The analysis combines a static and a dynamic analysis 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 outside of Garm's monitoring infrastructure. Garm introduces support for tracking provenance information across executions and application boundaries.

2. Provenance Analysis

Garm uses a staged provenance analysis that combines a static and dynamic analysis and operates on an application's binary. Garm instruments the application's binary through binary-rewriting using Valgrind. The system rewrites the binary when the binary executed. The code is rewritten on demand — only the parts of the program that actually executed are rewritten. Garm's analysis and instrumentation operate on Valgrind's intermediate representation, which abstracts many of the complexities of the x86 instruction set.

2.1 State

Garm maintains shadow state of an application's state. The shadow stores a 32-bit provenance value that describes a byte's current provenance. The application's state consist of its memory, the processor's registers, and any files that the application accesses:

- **Shadow Memory:** Garm contains a two-level table to shadow an application's memory: 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. When the application writes to an address, it allocates the second level table if necessary.
- **Registers:** Valgrind maintains the guest 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.
- **Temporaries:** Valgrind's intermediate representation introduces several temporary variables. Garm traces the provenance of temporary variables using a single shadow provenance regardless of the temporary'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 provenance of each byte in the original file.

2.2 Optimizing Instrumentation

The algorithm uses a data flow analysis to compute the static provenance of the registers and temporaries at each instruction. A static provenance is a set of instructions, register locations, temporary variables, and static list of instructions. The analysis uses this information to remove redundant provenance computations (i.e. the provenance computation for `d` in `b=a+c; d=b+c;` is redundant and can simply copy the provenance of `b`), and compute the provenance contribution from the execution of code. This analysis is structured as a standard dataflow analysis over static provenance.

2.3 Instrumenting Sources

We next describe how the instrumentation algorithm processes statements that may be a source for a provenance computation. When the instrumentation algorithm visits a load statement that should be instrumented, it generates a shadow load statement that loads the corresponding provenance from the shadow memory. When the algo-

rithm processes a register load statement, it checks to see if the register offset should be instrumented 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 into a temporary.

2.4 Computing Provenance

We next describe how the algorithm translates a static provenance from the static analysis into code that computes the corresponding dynamic provenance value. 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 instructions. These provenance 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 provenance 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.

The shadow temporaries provide a further opportunity for optimization. The provenance of shadow temporaries may be a subset of the current provenance. In this case, the instrumentation code for these sources is elided in the computation of the current temporary's provenance. This optimization computes the set of sources for the temporaries (and their shadow temporaries) and elides these sources from the current instrumentation.

2.5 Storing Provenance

The algorithm instruments statements that write values to memory or write values to registers that may be visible outside of the superblock. The algorithm instruments memory stores using the algorithm from Section 2.4 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.

2.6 Provenance Representation

This section presents Garm's provenance representation.

2.6.1 Base Provenance

Garm uses *base provenance* records to track the sources of incoming data. The first time that any application under Garm accesses data from a given file, Garm records a base provenance record that contains the file name, the current execution number, and a flag that denotes that this is the first time the Garm has seen this data. If an application under Garm accesses data that was modified by a previous application under Garm, Garm records 1) the unique execution identifier[1] for that previous execution, 2) the file name, 3) a reference to the 32-bit composite provenance value from the previous execution, and 4) a list of references to all access policies that apply. Garm represents base provenance records 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 provenance values have the same index.

2.6.2 Composite Provenance

An application's execution 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' provenance. Garm uses *composite provenance* values to track the provenance of a program's state. A composite provenance for a byte contains references to the base provenance records that contributed to the creation of the byte. Garm also records the list of references to the policies that apply to the composite provenance for efficiency reasons. Garm represents composite provenance as a 32-bit index into the composite provenance table. Each entry in the composite provenance table lists the component base provenance and any applicable data policies. Garm maintains the invariant that two identical composite provenance values must have the same index.

A byte's provenance can be viewed as an acyclic graph. Each composite provenance can reference several base provenance records and each base provenance can reference a composite provenance record from a previous execution. Garm contains a query tool that allow users to explore the provenance graph of an application's output.

2.6.3 Merging Provenance

The primary operation that the instrumented code performs is merging multiple source composite provenance values into a single output composite provenance. We next describe the basic algorithm for merging two provenance values, we construct the procedure for merging more than two provenance values from multiple invocations of two provenance values merge procedure:

1. **Identity Check:** The merge procedure first checks if the input composite provenance indices are the same and if so simply returns that index.
2. **Merge Cache Lookup:** The merge procedure next performs a hash table lookup on the two input composite provenance indices to see if the merge result 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 provenance as sorted lists of references to base provenance values and policies, it can merge two composite provenance records in a single pass. 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

[1] Garm assigns every program execution a unique execution identifier.

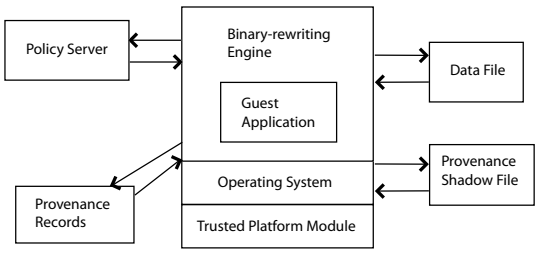

Figure 1. System Overview

the new composite provenance to the table and generates a new composite provenance index. Otherwise, it uses the composite provenance index from the table. Finally, it caches the results of the merge operation in the merge cache — it stores the input composite provenance indices and the output index.

3. System Architecture

We next describe Garm's basic architecture. Figure 1 presents an overview of Garm. The system is centered around the binary rewriting engine. The rewriting engine combines static and dynamic analyses to trace the provenance of the data that the application manipulates.

The rewriting system intercepts the system calls that the guest application performs. When the guest application loads data from the operating system, the binary rewriting engine loads the corresponding provenance information from the provenance shadow file and copies the provenance information to the shadow memory.

Policy servers are managed by the party who originally protected the data or on their behalf. Policy servers are responsible for authenticating Garm instances, determining whether they have permission to obtain keys, and transferring keys. If the provenance information indicates that the data is protected by a policy, Garm sends the encrypted policy information to the policy server along with information about the application and the user's identity. Garm conceptually uses the remote attestation capability of the trusted platform module to enable the policy server to verify that neither the underlying operating system nor Garm have been tampered with. The policy server verifies that Garm has not been tampered by verifying the remote attestation. Then the policy server decrypts 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 key to Garm. Garm stores the policy and key in memory during the current execution. 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 information for the data that was written.

3.1 Trusted Computing

Garm assumes that the underlying operating system and hardware supports trusted computing. Many trusted computing systems provide support for remote attestation. Remote attestation enables an application to prove both its identity and that it has not been tampered with to remote systems. Garm would use remote attestation to prove to the policy server that Garm has not been tampered with. 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 would use sealing to secure the private keys that it uses to sign the provenance shadow files to secure them against tampering as described in Section 3.3.

3.2 Protecting Data

Garm ensures that an application's execution respects the data policies of the data it accesses. When the application writes policy-protected data to a file, Garm encrypts the data to ensure that applications outside of the Garm framework cannot access the protected data.

3.2.1 Cipher Choice

Making encryption transparent to the application and enforcing access policies at a fine granularity constrains 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 the Salsa20 stream cipher as it satisfies both requirements [1]. Salsa20 was one of the stream ciphers selected by the ECRYPT Stream Cipher Project. 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 string which is xor'ed with the plaintext to encrypt the data or xor'ed with the ciphertext to decrypt the data. Garm associates a nonce with each file and policy pair. Notably, Salsa supports decrypting or encrypting blocks in files without requiring processing the previous blocks.

3.2.2 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 $message_2 \oplus keystream$. 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. Moreover, an attacker can exploit redundancy in the messages to obtain both plaintexts. It is therefore imperative that a given key and nonce is used at most once to write to a given file location. To address this weakness, Garm would monitor the locations a given nonce has been used to encrypt. If a location is repeated, Garm would generate a new nonce for the given policy and file. Garm can either re-encrypt the data that uses the current nonce, or it can assign a special provenance value for a 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.

3.3 Authentication of Provenance Data

If Garm is used to trace provenance, 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.

4. Experience

We next discuss our experience using Garm to trace provenance and enforce data access policies with several applications. We have developed a prototype implementation of Garm. Our prototype implements the provenance analysis, the stream cipher, the policy server, and a limited set of policies. These policies are: access once, unencrypted output to the audio device, unencrypted output to the screen, unencrypted output to any source, and encrypted only output. While it is straightforward, our prototype does not interface with the operating system to support remote attestation or sealing keys.

4.1 Data Provenance

We first discuss our experience using Garm to trace provenance 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, sort, and gcc. After each execution, we used the provenance viewing tool to examine the provenance of the output.

Text Editors: We first discuss our experiences using vi and nano, two interactive text editors. We edited a single file in several vi and nano sessions. Afterward, we viewed the provenance of the characters in the text file using the provenance viewing tool and verified its 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.

Gzip and Tar: We used gzip to compress and decompress the same text file. We then examined the provenance of the decompressed file. We observed that they were conservative. However, we did observe some imprecision in the provenance introduced in the process — the provenance of a few bytes included extra editing sessions. We then used tar to archive several files and then decompressed the archive. Garm was able to conservatively and precisely trace provenance across the two tar executions.

Sort: We used sort to sort a text file developed over several sessions. The results were conservative. We also observed mixing of provenance, but in this case the location of a text line depends on the other lines and the mixing of provenance simply reflects this dependence.

GCC: Finally, we used gcc to compile several source files into a binary. One invocation of gcc runs several program invocations including a compiler, an assembler, and a linker. Garm was able to trace provenance across all of these executions. We then examined the provenance for text strings in the program's binary and found that they were accurate.

4.2 Policy Enforcement

Our Garm prototype supports a limited set of policies including combinations of access once, allow viewing 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.

mpg123: We used Garm with mpg123, an MP3 player, to play a protected MP3 file. We explored data policies that explored many combinations of the basic access policies. For example, we explored MP3 files that could be played once, MP3 files that could be played but whose ID3 information could not be displayed to the screen, and MP3 files whose ID3 information was displayed but that could not be played through the audio device. We found that Garm successfully enforced these policies and that Garm had sufficient performance to run mpg123 in real-time.

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

GCC: 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 policy from that binary and then verified that the binary executed correctly.

5. Related Work

Taint analysis tracks whether an application's data is tainted. Tainting can be used to represent that the information is secret or that the information is from an untrusted source. Researchers have developed many taint analyses [8, 7]. Taint analyses are very coarse-grained and typically allocate only 1 bit per value. Therefore tainting often makes the implicit assumption that the user knows *a priori*, which sources of values should be tainted. Garm supports a rich set of provenance that describe the history of how the data was generated and therefore does not require that the user identify tainted sources ahead of time.

Most current taint frameworks enforce policies at the boundary of the tainting framework [6]. 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 tracking 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 its widespread adoption may ultimately depend on not burdening users with onerous restrictions on how they use data.

The HiStar operating system provides information flow control capabilities [9]. It requires the user to annotate each application with a set of permissions and a set of privileges. Garm's policies instead focus on what applications are allowed to do with the data at the interface to the operating system.

PinUP can enforce policies on how applications use files at the file granularity [3]. For example, a policy might only allow Word to access documents. This approach can restrict normal processes such as emailing documents and fails to differentiate between confidential and public documents of the same type.

Researchers have studied the problem of tracking and maintaining provenance in databases [2]. Garm presents a technique that can track provenance on arbitrary binaries at a level of abstraction that captures sufficient information to easily track the source while simultaneously not incurring excessive overheads. Researchers have developed automated provenance gathering frameworks that operate at the file granularity [5] — for each outputted file they record the application that created the file, how it was invoked, and a list of all files it read. Researchers have developed library level tools to produce verifiable provenance records for files [4]. Our approach is much more precise — Garm can determine which of multiple inputs (at the byte granularity) contributed to a given output byte.

6. Conclusion

Currently, it is difficult to discover the history of the creation of a file or to protect data in the files we send to others. Garm uses a staged analysis to track 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

[1] D. J. Bernstein. *New Stream Cipher Designs: The eSTREAM Finalists*, chapter The Salsa20 Family of Stream Ciphers, pages 84–97. Springer, 2008.

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

[3] W. Enck, P. McDaniel, and T. Jaeger. PinUP: Pinning user files to known applications. In *Proceedings of the Annual Computer Security Applications Conference*, 2008.

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

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

[6] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engingeering and Methodology*, 9(4):410–442, 2000.

[7] N. Vachharajani et al. RIFLE: An architectural framework for user-centric information-flow security. In *the Annual International Symposium on Microarchitecture*, 2004.

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

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