Cryptographic Enforcement of Language-Based Information Erasure

Aslan Askarov
Aarhus University

Scott Moore
Harvard University

Christos Dimoulas
Harvard University

Stephen Chong
Harvard University

***Abstract*—Information erasure is a formal security requirement that stipulates when sensitive data must be removed from computer systems. In a system that correctly enforces erasure requirements, an attacker who observes the system after sensitive data is required to have been erased cannot deduce anything about the data. Practical obstacles to enforcing information erasure include: (1) correctly determining which data requires erasure; and (2) reliably deleting potentially large volumes of data, despite untrustworthy storage services.**

In this paper, we present a novel formalization of language-based information erasure that supports cryptographic enforcement of erasure requirements: sensitive data is encrypted before storage, and upon erasure, only a relatively small set of decryption keys needs to be deleted. This cryptographic technique has been used by a number of systems that implement data deletion to allow the use of untrustworthy storage services. However, these systems provide no support to correctly determine which data requires erasure, nor have the formal semantic properties of these systems been explained or proven to hold. We address these shortcomings. Specifically, we study a programming language extended with primitives for public-key cryptography, and demonstrate how information-flow control mechanisms can automatically track data that requires erasure and provably enforce erasure requirements even when programs employ cryptographic techniques for erasure.

I. Introduction

The security requirements of modern software often impose restrictions on the lifetime of sensitive information. For instance, to protect its users, electronic payment software should erase credit-card information after the transaction. Similarly, browsers running in "private mode" promise (but often fail) to not leak or permanently store any information after the end of a private session [3, 35, 43].

These examples reveal two significant challenges for programs whose security depends on erasing sensitive information when a dynamic condition is met. The first challenge is to track sensitive data in a program in order to erase it at the appropriate time. This is complicated by the need to track data *derived from* sensitive data, and erase that too. For example, a browser in private mode must delete cookies from visited websites but also delete history, passwords, text box input and other cached information that may reveal user activity from a private session. Hence, to correctly erase sensitive information, it is necessary to track and control information dependencies; a difficult task for complex programs.

The second challenge is that applications that handle sensitive information often receive, generate, and store large amounts of data: so much data that the application must use large but untrustworthy storage services (such as the cloud, or hard drives, depending on the trust model). For example, the back-end of a high-traffic credit-card processing service handles millions of requests and generates terabytes of logs every day. Untrustworthy storage services may not be able to delete data reliably or quickly.

In this paper, we address these two challenges by providing language-level support to track sensitive information that requires erasure and use cryptographic mechanisms to efficiently and reliably delete data when dynamic conditions are met. Previous work considers these challenges in isolation; we bridge the gap between these separate lines of work and address both challenges. We improve on previous language-based techniques for information erasure by making it more practical, while clarifying and extending the guarantees offered by existing systems that use cryptography for data erasure.

Motivating example. Snapchat is a photo messaging smartphone app that allows senders to specify a time limit after which a message should be erased from the recipient's device. Snapchat did not correctly erase messages, allowing recipients to access supposedly erased messages via the file system [25] and resulting in a complaint filed with the FTC [23].

Consider a Snapchat-like application that receives sensitive data that should be erased under certain conditions (e.g., after a fixed period of time, at the user's command, or when a certain event happens, such as 10 consecutive incorrect password attempts). The data is stored in an untrustworthy file system (which is untrustworthy because its contents may be viewed by untrusted parties) and deletion of data from the file system may not be immediate or permanent [10, 11, 20, 21, 22].[1] When the application computes on sensitive data, for example by allowing a user to manipulate sensitive documents, it produces additional sensitive data that should also be erased when required.

Note that this simple application has the two challenges described above: there is sensitive information that needs to be erased, and the application relies on an untrustworthy storage service to store data that may need to be erased. Consider how this application could be implemented using existing approaches: language-based information-flow control and cryptographic data deletion.

[1]This reflects the Snapchat scenario, where a smartphone user could access Snapchat messages via the device's file system, which does not reliably delete data. It also applies to applications that use cloud storage services.

Appeared in CSF 2015

Language-based information-flow control provides formal foundations for information erasure [12] and its provably correct enforcement through language techniques, such as security-type systems [13, 27] that track sensitive data. If our Snapchat-like application is written in a language that tracks and controls information flow to enforce information erasure, then the programmer can express the application's erasure requirements as security policies, and language mechanisms can ensure that the application correctly tracks all data that needs to be erased. However, the application's storage service is untrustworthy, and this approach is unable to ensure that the storage service correctly deletes data, even though the data that should be deleted is accurately identified.

Alternatively, our application could be written in a language or system that supports cryptographic data deletion. These systems allow sensitive information to be stored encrypted in untrusted storage services and erase information by erasing the encryption keys [10, 42, 45]. Following this approach, our Snapchat-like application would encrypt the sensitive data before storing it, and would "delete" the stored data by deleting the encryption keys. Thus, the application avoids trusting the untrustworthy storage service. The application does still require a trusted store for storing and reliably deleting cryptographic keys, but the requirements and scale of this trusted store are simpler than the (file system or cloud) storage service.

However, this approach provides no support for ensuring that information derived from sensitive data is correctly erased. The developer must track information dependencies to ensure all derived information is deleted appropriately. This is an error-prone manual process, leading to security vulnerabilities. Moreover, such systems lack formal foundations for the security guarantees they are attempting to provide, and so it is unclear whether they handle sensitive information correctly.

In this work, we combine the two approaches while avoiding the shortcomings of each. Since our goal is a practical, yet provably correct, cryptographic enforcement mechanism for information erasure, it is important to provide both expressive erasure policies and a clear semantics for their correct cryptographic enforcement. This requires the development of a formal framework that can be extended with cryptographic primitives without obscuring the semantics of information erasure. At its core is a novel, simple and intuitive semantic definition for information erasure that supports the cryptographic enforcement of erasure. Previous work has either achieved the same level of semantic simplicity by sacrificing the expressiveness of erasure policies [27] or sacrificed simplicity and thus easy extensibility of the semantics of erasure [12]. Our framework manages to meet both requirements: expressive dynamic erasure policies and a simple extensible semantic definition of erasure.

In our approach, security policies describe the current confidentiality level of data, when to erase it, and the security level to enforce on the data after erasure. Often the security level to enforce after erasure will indicate that the data should be removed from the system, but our framework can handle more general requirements where the data is made more restricted, but does not need to be removed.

For example, a suitable policy in our SnapChat-like application for a document that needs to be erased upon multiple incorrect password guesses would be $\mathtt{L}^{\mathsf{10BadGuesses}\nearrow\top}$. Here, L is a low-confidentiality security level, $\top$ is the top security level and 10BadGuesses is a program variable that the application sets upon ten consecutive incorrect password guesses. Intuitively, if this policy is associated with data, it requires that the data may initially be treated in accordance with security level L, but if and when variable 10BadGuesses is set, then the data (and any data derived from it) must have security level $\top$ enforced on it, requiring the removal of the data from the system.

We enforce erasure policies with a flow-sensitive type and effect system. To gain insight into how our enforcement mechanism works, consider our Snapchat-like application. The untrusted store (i.e., the file system or cloud storage service) is permitted to store only information at level L or lower (enforced by the type system restricting calls to the store's API). Since the sensitive document has policy $\mathtt{L}^{\mathsf{10BadGuesses}\nearrow\top}$, which is *not* lower than L in our security policy ordering, the document could not be stored in the untrusted store. However, given an appropriate cryptographic key, the document could be encrypted, and the resulting ciphertext (which has level L) could be stored in the untrusted store. An appropriate cryptographic key is a key that will be erased when variable 10BadGuesses is set. Our type system ensures that only appropriate keys are used and that the keys themselves are handled appropriately. Using standard information-flow control techniques, the type system also ensures that appropriate policies are enforced on data derived from sensitive data.

We have developed a tool, KEYRASE, which extends our enforcement mechanism to the Java programming language, and used it to implement a chat client that supports information erasure of received messages stored in an encrypted log. This demonstrates the practicality and usefulness of our approach to express and enforce realistic information erasure requirements.

This paper makes the following three contributions:

1) A novel semantic security condition for information erasure. The security condition is easily extended with cryptographic primitives, providing an intuitive foundation for cryptographic information erasure and its correct enforcement. Moreover, it formalizes and justifies existing mechanisms for cryptographic data deletion.

2) A flow-sensitive type and effect system that provably enforces information erasure for an imperative language with cryptographic primitives. This is the first such cryptographic enforcement mechanism for dynamic erasure policies. Proving correctness requires showing the type and effect system is a sound and accurate static approximation of the dynamic erasure policies, a task significantly more complex than standard noninterference-like proofs. However, the complexity of the proof does not pollute the simplicity of the semantic definition or the type system.

3) An implementation of the enforcement mechanism for Java (including an extension for erasing information from

$$
\begin{aligned}
c &::= \mathsf{skip} \mid x := e \mid c_0; c_1 \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
&\quad \mid \mathsf{set}(x) \mid \mathsf{init}\ \underline{y}\ \mathsf{to}\ e \mid \mathsf{read}\ \underline{y}\ \mathsf{into}\ x \mid \mathsf{output}(\ell, e) \\
e &::= v \mid x \mid e_1\ \mathsf{op}\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_i(e) \\
v &::= n \mid (v, v) \\
x &\in \mathbf{Var},\ \underline{y} \in \underline{\mathbf{Var}},\ \ell \in \mathcal{L}
\end{aligned}
$$

Fig. 1. Simple imperative language

the heap). The implementation, together with a chat client that supports erasing information from previously received messages, provides evidence for the pragmatic value of our approach and its applicability to real languages.

Note that in this work we are concerned with "logical erasure" [16], not physical erasure [21, 22]. We do not address the problem of reliably erasing data from physical media. Also, we focus on confidentiality guarantees, and assume the attacker can observe but not modify data in untrustworthy storage.

II. Language, attacker model, and security

This section presents a novel security definition for information erasure. We first describe a language model and use it to express an appropriate attacker model.

In our model, deletion of data corresponds to an assignment to the variable containing the data. For locations the program cannot reliably delete, we use *write-once variables*, which can be initialized but not overwritten. For example, the untrustworthy storage of our Snapchat-like example corresponds to one or more write-once variables. The identifier of a write-once variable is underlined (e.g., $\underline{y}$, $\underline{z}$).

We assume a set of security levels $\mathcal{L}$, and use meta-variable ℓ to range over security levels. Security levels describe confidentiality requirements, and we assume a partial order $\sqsubseteq$ that describes the relative restrictiveness of security levels. If $\ell \sqsubseteq \ell'$ then security level ℓ' is at least as restrictive as security level ℓ, i.e., ℓ' requires at least as much confidentiality as ℓ. We assume the partial order has a greatest element, denoted $\top$. In our examples, we use the two-point security lattice with elements $\mathtt{L}$ and $\mathtt{H}$, where $\mathtt{L} \sqsubseteq \mathtt{H}$, but $\mathtt{H} \not\sqsubseteq \mathtt{L}$ (i.e., $\mathtt{H} = \top$).

Programs interact with the external environment via communication channels. For simplicity, we assume there is exactly one communication channel per security level.

A. Language and attacker model

Figure 1 depicts the syntax of our language. The commands of the language are standard with the exception of set, init, read, and output. Command set sets variables that are used as erasure conditions; we discuss it further in Section II-B. Command init $\underline{y}$ to e creates a write-once variable $\underline{y}$ and initializes it to the result of e; attempts to re-initialize a write-once variable block program execution. Command read $\underline{y}$ into x assigns the value stored in $\underline{y}$ to x. Command $\mathsf{output}(\ell, e)$ outputs the value of e on channel ℓ. Outputs correspond to attacker observations.

For evaluation of commands, we use a small-step semantics. A configuration is a tuple $\langle c, m, \mathsf{U}, t\rangle$, where c is a command, m is a standard memory that maps standard variables to values, U is a write-once memory that maps write-once variables to values, and t is an output trace. Output traces have the form $(\ell_1, v_1) \cdot \ldots \cdot (\ell_n, v_n)$, where (ℓ_i, v_i) records that the ith output was of value v_i on channel ℓ_i. We write $t \cdot t'$ for concatenation of traces t and t', and ε for the empty trace. The initial configuration for program c with initial memory m is $\langle c, m, \mathsf{U}_{\mathsf{init}}, \varepsilon\rangle$, where $\mathsf{U}_{\mathsf{init}}$ is the initial write-once memory (i.e., the one where no write-once variable has been written to). The reduction relation $\rightarrow$ for configurations is straightforward and we omit it. Note that the semantics enforces that write-once variables are written to at most once, and that $\mathsf{set}(x)$ is semantically equivalent to assigning 1 to x. We use a large step evaluation relation for expressions; $\langle m, e\rangle \Downarrow v$.

The following program models the interaction of our Snapchat-like example with the phone display and an untrusted store. We use a write-once variable to model the untrusted store, since it is not trusted to reliably delete data. A standard variable represents the display, since the program can erase it by displaying new data. Observations of the display by the user are modeled by output statements. In this program msg contains sensitive information that should be erased by the time condition variable $delete$ is set.

```
1  display := msg; output(L, display);     // Display message
2  compressed := gzip(msg);                // Compress message
3  init store to compressed;               // Store message
4  display := 0; msg := 0; compressed := 0; set(delete);
5  output(L, display);
6  read store into tmp; output(L, tmp);    //Observe store
```

The program displays the message on the screen (line 1), and compresses it using $gzip$ (line 2), and then writes the compressed message to the storage device (line 3).[2] Importantly, even though the compressed message differs from msg, it carries information that depends on msg. That is, there is information flow from msg to the result of $gzip(msg)$.

Finally, the program attempts to delete the message by overwriting $display$, msg, and $compressed$ (line 4) just before $delete$ is set. However, it cannot overwrite the write-once variable $\underline{store}$. Later, an attacker may be able to observe its contents (line 6), and thus learn the sensitive message after it should have been deleted. This program does *not* correctly erase all information about the sensitive message.

Attacker observations and knowledge. The previous example appeals to an intuitive notion of secure information erasure. To turn this notion into a formal definition, we develop a formal attacker model.

An attacker is an entity that observes the outputs performed on some set of communication channels during some contiguous period of program execution. Based on these observations, the attacker infers facts about sensitive data contained in the initial memory.[3] We assume an attacker observes only some of

[2] Our model does not include functions but we write $gzip(msg)$ to abstract the process of compressing msg.

[3] We assume sensitive information comes from the initial memory; orthogonal generalizations to input streams [4, 6] or user strategies [39] are possible.

a program's execution because we are interested in whether an attacker that starts observing after sensitive information should be erased learns anything about the sensitive information.

We associate each attacker with a security level ℓ, and assume that an attacker with security level ℓ can observe outputs on all channels ℓ' such that $\ell' \sqsubseteq \ell$. Given a subtrace t, which represents the output of a program during some contiguous part of its execution, the projection of t to level ℓ, $\lfloor t \rfloor_\ell$, describes the observations of an attacker with security level ℓ; it is the list of outputs in t on any channel ℓ' such that $\ell' \sqsubseteq \ell$. Suppose execution of a program produces trace $t \cdot t_{\mathsf{obs}}$ and the attacker observes t_{obs}. The knowledge of the attacker is the set of initial standard memories that are consistent with these observations. An initial memory is consistent with an attacker's observation if execution of the program under that memory could produce trace $t' \cdot t''$ such that $\lfloor t'' \rfloor_\ell = \lfloor t_{\mathsf{obs}} \rfloor_\ell$.

Definition 1 (Attacker knowledge)**.** *Given a program c, security level ℓ, and subtrace t_{obs} observed by the attacker, define* attacker knowledge *at level ℓ for subtrace t_{obs} as follows*

$$k_\ell(c, t_{\mathsf{obs}}) = \{m \mid \langle c, m, \mathsf{U}_{\mathsf{init}}, \varepsilon\rangle \longrightarrow^* \langle c', m', \mathsf{U}', t_1\rangle \longrightarrow^* \langle c'', m'', \mathsf{U}'', t_1 \cdot t'\rangle \wedge \lfloor t_{\mathsf{obs}} \rfloor_\ell = \lfloor t' \rfloor_\ell\}$$

The smaller the attacker knowledge set, the more the attacker knows. For example, an attacker that makes no observations (i.e., $\lfloor t_{\mathsf{obs}} \rfloor_\ell = \varepsilon$) has all initial memories in its attacker knowledge, meaning that the attacker has no information about the initial memory of the execution.

As an example of attacker knowledge, assume that the initial standard memory contains two variables x and y, and we execute the program $\mathsf{output}(\mathtt{L}, x); \mathsf{output}(\mathtt{L}, y)$. Now assume that the attacker's observation is $\lfloor t_{\mathsf{obs}} \rfloor_{\mathtt{L}} = (\mathtt{L}, 1)$. Either of the two commands may have produced this output. So the attacker knows that either the initial value of x is 1 or the initial value of y is 1. Therefore, the knowledge of this attacker is $\{m \mid m(x) = 1 \vee m(y) = 1\}$.

A similar scenario occurs in the Snapchat-like example from this section. Assume we execute the program with an initial memory m where $m(msg) = 42$, and an attacker at level L observes only the output on line 5. The attacker observes only the trace $(\mathtt{L}, 0)$. All initial memories are consistent with this observation, so the attacker learns nothing; the attacker's knowledge is the set of all memories. However, if the attacker also observes the output on line 6 then the attacker's observations are $(\mathtt{L}, 0), (\mathtt{L}, \mathit{gzip}(42))$, and the attacker's knowledge is $\{m \mid m(msg) = 42\}$.

B. Erasure policies and erasure environments

So far, we have defined erasure policies informally. In order to express these security requirements formally, we specify the syntax of erasure policies:

$$p, q ::= \ell \mid p \overset{cnd}{\nearrow} q$$

An erasure policy is either a security level ℓ or a policy of the form $p \overset{cnd}{\nearrow} q$, where cnd is the erasure condition for this policy. Intuitively, information with erasure policy $p \overset{cnd}{\nearrow} q$ is treated according to policy p until condition cnd is set, and thereafter must be treated according to policy q. Conditions cnd are program variables. We assume that initially all condition variables are set to 0. In Section IV, we give these variables the type cond, which prevents assignments to condition variables. Condition variables can be set only by the command set, which updates their value to 1. A simple security level ℓ means that no erasure is required, and the information has security level ℓ for the duration of program execution. Condition variables provides a simple but expressive way for a programmer to connect erasure conditions to program semantics. We conjecture that more general erasure conditions, such as arbitrary program expressions or semantic conditions [12, 13], do not significantly improve the practicality of the model.

We attach erasure policies to sensitive data through an *erasure environment*. An erasure environment $\gamma : \mathbf{Var} \to \mathbf{Pol}$ maps variables to erasure policies. Given erasure environment γ and variable x, $\gamma(x)$ is the erasure policy that should currently be enforced for information derived from the initial contents of x, i.e., the value of x in the initial memory.

The initial erasure environment of a program serves as the specification of its erasure requirements. For instance, in the Snapchat-like example the initial erasure environment maps msg to policy $\mathtt{L} \overset{delete}{\nearrow} \mathtt{H}$. The erasure environment may change during program execution when the program sets condition variables. For example, after $delete$ is set during the execution of our example program, the erasure environment changes to map msg to policy H: the initial value of msg and data derived from it should now be protected at level H. To capture the dynamic nature of erasure requirements, we augment configurations with erasure environments, $\langle c, m, \mathsf{U}, t\rangle_\gamma$. We lift the reduction relation using the following rule.[4]

$$\textsf{pol-env-update}\quad \frac{\langle c, m, \mathsf{U}, t\rangle \to \langle c', m', \mathsf{U}', t'\rangle \qquad \gamma' = \mathsf{update}(m', \gamma)}{\langle c, m, \mathsf{U}, t\rangle_\gamma \to \langle c', m', \mathsf{U}', t'\rangle_{\gamma'}}$$

The rule uses update to recursively update each policy in γ when a condition cnd is set. Function update takes two arguments: the memory after setting cnd and the erasure environment, and returns the updated erasure environment.

$$\mathsf{update}(m, \gamma) = \lambda x \,.\, \mathsf{upd}(m, \gamma(x))$$

$$\mathsf{upd}(m, p) = \begin{cases} \ell & \text{if } p = \ell \\ \mathsf{upd}(m, q) & \text{if } p = p' \overset{cnd}{\nearrow} q \\ & \text{and } m(cnd) \neq 0 \\ \mathsf{upd}(m, p') \overset{cnd}{\nearrow} \mathsf{upd}(m, q) & \text{if } p = p' \overset{cnd}{\nearrow} q \\ & \text{and } m(cnd) = 0 \end{cases}$$

C. Security for erasure

Using attacker knowledge and erasure environments, we can define what it means for a program to erase information securely. Intuitively, a program is secure iff at any point in its execution, it produces only observable effects that are consistent with the security requirements of the program at the

[4]Notice that γ-lifting does not change the behavior of programs.

moment of the effect. This accounts for changes of erasure policies during execution. In more concrete terms, a program is secure iff it outputs information on channel ℓ only when the currently enforced *security level* for that information permits it to flow to channel ℓ. Function $\mathsf{cur}(p)$ returns the current security level of p.

$$\mathsf{cur}(p) = \begin{cases} \ell & \text{if } p = \ell \\ \mathsf{cur}(p') & \text{if } p = p' \overset{c}{\nearrow} q \end{cases}$$

We use this function to describe what information an attacker at level ℓ can learn given an erasure environment γ. Specifically, for any variable x and initial memory m, the attacker is permitted to learn about the initial value of x only if the currently enforced security policy for $m(x)$ (i.e., $\mathsf{cur}(\gamma(x))$) is allowed to flow to ℓ. The following definition expresses this idea as a set of memories that agree on the contents of x:

Definition 2 (Indistinguishable memories)**.** *Given memory m, erasure environment γ, and security level ℓ, define* indistinguishable memories according to γ at ℓ *to be the set*

$$\mathsf{ind}_\ell(m,\gamma) \triangleq \{m' \mid \forall x\,.\,\mathsf{cur}(\gamma(x)) \sqsubseteq \ell \implies m(x) = m'(x)\}$$

We can use indistinguishable memories to define what information an attacker at level ℓ who observes a subtrace t_{obs} of the execution is permitted to learn: the intersection of all the initial memories that are consistent with the erasure policies that were current during the production of t_{obs}. This means that if the erasure environment changes while an attacker observes the program, the information the attacker is permitted to learn includes information both before and after the change. For instance, if the execution of the program with initial memory m involves at least n configurations and the attacker observes a subtrace that corresponds to configurations i to n, the attacker is allowed to learn $\bigcap_{i\le j\le n} \mathsf{ind}_\ell(m,\gamma_j)$. Since (i) a program is secure iff the information an attacker learns is bound by the information the attacker is allowed to learn, and (ii) attacker knowledge (Definition 1) captures the information an attacker learns, we define security as follows:

Definition 3 (Security)**.** *Program c is* secure *for initial erasure environment γ if for all memories m and executions $\langle c, m, \mathsf{U}_{\mathsf{init}}, \varepsilon\rangle_\gamma \longrightarrow^* \langle c_i, m_i, \mathsf{U}_i, t_i\rangle_{\gamma_i} \longrightarrow^* \langle c_n, m_n, \mathsf{U}_n, t_i \cdot t_{\mathsf{obs}}\rangle_{\gamma_n}$ where $i \le n$, it holds that, for all levels ℓ, we have $k_\ell(c, t_{\mathsf{obs}}) \supseteq \bigcap_{i\le j\le n} \mathsf{ind}_\ell(m,\gamma_j)$*

Note that an attacker who observes a subtrace of a program's execution is allowed to learn less than an attacker who observes a larger subtrace. Conversely, attackers observing a larger part of the computation may obtain more precise knowledge. Also note that this definition of security is progress sensitive: it rejects programs that leak sensitive information via their termination behavior [8]. Finally, when erasure environments do not change during execution, i.e., $\gamma = \gamma_i$, for all $i \le n$, the definition coincides with non-interference.

We conclude this section with an example. Consider a two-point security lattice with elements $\mathtt{L}$ and $\mathtt{H}$, an erasure condition variable t, variable x, erasure environment γ such that $\gamma(t) = \mathtt{L}$ and $\gamma(x) = \mathtt{L} \overset{t}{\nearrow} \mathtt{H}$ and program $\mathsf{output}(\mathtt{L}, x);\ \mathsf{set}(t);\ \mathsf{output}(\mathtt{L}, x)$. This program outputs the value of x after condition variable t is set. It is insecure and is rightfully rejected by Definition 3. If $m(x) = 42$ and $m(t) = 0$ in the initial memory, the program execution is as follows:

$$\begin{aligned} &\langle \mathsf{output}(\mathtt{L}, x);\ \mathsf{set}(t);\ \mathsf{output}(\mathtt{L}, x), m, \mathsf{U}_{\mathsf{init}}, \varepsilon\rangle_\gamma \longrightarrow \\ &\quad \langle \mathsf{set}(t);\ \mathsf{output}(\mathtt{L}, x), m, \mathsf{U}_{\mathsf{init}}, (\mathtt{L}, 42)\rangle_\gamma \longrightarrow \\ &\quad\quad \langle \mathsf{output}(\mathtt{L}, x), m', \mathsf{U}_{\mathsf{init}}, (\mathtt{L}, 42)\rangle_{\gamma'} \longrightarrow \\ &\quad\quad\quad \langle \mathsf{stop}, m', \mathsf{U}_{\mathsf{init}}, (\mathtt{L}, 42) : (\mathtt{L}, 42)\rangle_{\gamma'} \end{aligned}$$

Here $m' = m[t \mapsto 1]$ and $\gamma' = \mathsf{update}(m', \gamma) = \gamma[x \mapsto \mathtt{H}]$. For an attacker that observes the subtrace $t_{\mathsf{obs}} = (\mathtt{L}, 42)$ attacker knowledge is $k_{\mathtt{L}}(c, (\mathtt{L}, 42)) = \{m \mid m(x) = 42 \wedge m(t) = 0\}$.[5] In order to determine whether the attacker's knowledge is bounded by the information the attacker is permitted to learn, we must consider the sets of indistinguishable memories permitted by the configurations that produced t_{obs}. Our example program is secure (Definition 3) if $\mathsf{ind}_{\mathtt{L}}(m, \gamma') \subseteq k_{\mathtt{L}}(c, (\mathtt{L}, 42))$. Since $\mathsf{cur}(\gamma'(t)) = \mathtt{L}$ and $\mathsf{cur}(\gamma'(x)) = \mathtt{H} \not\sqsubseteq \mathtt{L}$, we have $\mathsf{ind}_{\mathtt{L}}(m, \gamma') = \{m \mid m(t) = 0\}$. Thus, the program is insecure, since $\{m \mid m(t) = 0\} \not\subseteq \{m \mid m(x) = 42 \wedge m(t) = 0\}$.

Similarly, our security definition rejects our Snapchat-like example. Recall that initially $m(msg) = 42$ and $\gamma(msg) = \mathtt{L} \overset{delete}{\nearrow} \mathtt{H}$. Assume that γ' is the erasure environment for the configuration that is about to perform the output on line 6. For this configuration, we obtain that $\gamma'(msg) = \mathtt{H}$ and $\mathsf{ind}_{\mathtt{L}}(m, \gamma') = \{m \mid m(delete) = 0\}$. However, for an attacker at level $\mathtt{L}$ who observes the output on line 6, attacker knowledge is $\{m \mid m(msg) = 42 \wedge m(delete) = 0\}$ which is not a subset of $\mathsf{ind}_{\mathtt{L}}(m, \gamma')$, and so the program is insecure.

III. Language with cryptographic primitives

In this section we extend our semantic security condition for erasure to a setting with cryptographic primitives. Our notion of symbolic cryptographic attacker is inspired by the one of Askarov and Sabelfeld [5]. but is different in that our extension is based on Laud's model for cryptographically-masked flows [31].

We use Laud's symbolic approach to cryptography because it has been proven sound for the computational cryptographic model. We do not directly prove computational soundness for this work, but such a proof would follow Laud's proof closely.

Section III-A extends our language with cryptographic primitives; technical development in that subsection is a direct adaptation of Laud's work [31, §8], and is not a contribution of this work. Section III-B extends our model of attacker knowledge to account for cryptographic operations, and correspondingly extends our semantic definition of security. We use a public key cryptographic model, although our development can be straightforwardly adapted for symmetric key cryptography. Specifically, in this setting, symmetric key cryptography is less challenging than public key cryptography. With public key cryptography, our security definitions and enforcement

[5]The initial value of all condition variables is known to be 0.

$$v ::= \ldots \mid \mathsf{pk}\langle i\rangle \mid \mathsf{sk}\langle i\rangle \mid \boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle} \mid \mathsf{NaV}$$
$$c ::= \ldots \mid x := \mathsf{encrypt}(e_1, e_2) \mid x := \mathsf{decrypt}(e_1, e_2) \mid \mathsf{newkeypair}(x_1, x_2, p)$$

Fig. 2. Language with cryptographic primitives

s-pub-encrypt

$$\frac{\langle m, e_1\rangle \Downarrow \mathsf{pk}\langle i\rangle \qquad \langle m, e_2\rangle \Downarrow v \qquad \mathsf{Y}(\mathsf{enc}) = j}{\langle x := \mathsf{encrypt}(e_1, e_2), m, \mathsf{U}, \mathsf{Y}, t\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto \boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle}], \mathsf{U}, \mathsf{Y}[\mathsf{enc} \mapsto j+1], t\rangle}$$

s-pub-decrypt

$$\frac{\langle m, e_1\rangle \Downarrow \mathsf{sk}\langle i\rangle \qquad \langle m, e_2\rangle \Downarrow \boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle}}{\langle x := \mathsf{decrypt}(e_1, e_2), m, \mathsf{U}, \mathsf{Y}, t\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto v], \mathsf{U}, \mathsf{Y}, t\rangle}$$

s-pub-decrypt-fail

$$\frac{\langle m, e_1\rangle \Downarrow \mathsf{sk}\langle i'\rangle \qquad \langle m, e_2\rangle \Downarrow \boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle} \qquad i \neq i'}{\langle x := \mathsf{decrypt}(e_1, e_2), m, \mathsf{U}, \mathsf{Y}, t\rangle \longrightarrow \langle \mathsf{stop}, m[x \mapsto \mathsf{NaV}], \mathsf{U}, \mathsf{Y}, t\rangle}$$

s-pub-newkey

$$\frac{\mathsf{Y}(\mathsf{pubkey}) = i}{\langle \mathsf{newkeypair}(x_1, x_2, p), m, \mathsf{U}, \mathsf{Y}, t\rangle \longrightarrow \langle \mathsf{stop}, m[x_1 \mapsto \mathsf{pk}\langle i\rangle, x_2 \mapsto \mathsf{sk}\langle i\rangle], \mathsf{U}, \mathsf{Y}[\mathsf{pubkey} \mapsto i{+}1], t\rangle}$$

Fig. 3. Semantic of cryptographic commands

mechanism must ensure that both public and private keys are used correctly, while with symmetric cryptography we would only have to consider symmetric keys. An adapted enforcement mechanism would impose similar restrictions on symmetric keys as it does on private keys. The model for encryption and decryption operations and ciphertexts would be similar as well.

A. Syntax and semantics

Figure 2 presents the extensions to our language syntax. We extend values with an indexed collection of key pairs, and refer to the individual keys as $\mathsf{pk}\langle i\rangle$ and $\mathsf{sk}\langle i\rangle$ for the public and private keys, respectively. We also use an indexed collection of random strings $\mathsf{r}\langle j\rangle$. These strings are not first-class values, but appear as components of encrypted values. Encrypted values have form $\boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle}$, and consist of a plaintext v, the indexed public key $\mathsf{pk}\langle i\rangle$ and the indexed random string $\mathsf{r}\langle j\rangle$ that was used for the encryption of v. We model randomized encryption, and $\mathsf{r}\langle j\rangle$ encapsulates the randomness used in the encryption. Including it in the symbolic model of ciphertexts allows us to model the fact that an attacker can distinguish between distinct ciphertexts even if the attacker cannot decrypt them.

Semantics. We extend configurations to have form $\langle c, m, \mathsf{U}, \mathsf{Y}, t\rangle$, where Y is the *symbolic cryptographic environment* that counts the number of key pairs and random strings generated in an execution. Y maps $\{\mathsf{pubkey}, \mathsf{enc}\}$ to $\mathbb{N}$. $\mathsf{Y}(\mathsf{pubkey})$ is the number of key pairs that have been generated during the execution so far, and $\mathsf{Y}(\mathsf{enc})$ is the number of encryptions that have been performed so far (and thus is equal to the number of random strings generated). The initial symbolic cryptographic environment, written $\mathsf{Y}_{\mathsf{init}}$, maps both counters $\mathsf{Y}(\mathsf{pubkey})$ and $\mathsf{Y}(\mathsf{enc})$ to 0. The counters are incremented by commands for key generation and encryption respectively.

Figure 3 presents operational semantics for the new commands. Inference rules for other commands are modified for the new configurations in the obvious way. Encryption command $x := \mathsf{encrypt}(e_1, e_2)$ evaluates e_1 to a public key, evaluates e_2 to a value v, encrypts v using the public key (generating a new random string to do so) and assigns the result to variable x. Decryption $x := \mathsf{decrypt}(e_1, e_2)$ evaluates e_1 to a private key and evaluates e_2 to a ciphertext. If the ciphertext was encrypted using the corresponding public key, the decryption succeeds, assigning to x the encrypted value. Otherwise, the decryption fails assigning to x the value NaV. Finally, $\mathsf{newkeypair}(x_1, x_2, p)$ generates a new key pair and assigns the public key to x_1 and the private key to x_2. Policy p is an annotation used by our enforcement mechanism, and is described in more detail in the following section. It is an upper bound on the information that the public key may be used to encrypt, and also restricts how the private key is handled.

Symbolic equivalence of encrypted values. In order to define what values an attacker can and cannot distinguish, we define an equivalence relation over ciphertexts. For this, we define a few auxiliary concepts based on a Dolev-Yao-style attacker model [17].

First, given a set of values $\mathbf{V}$, the *Dolev-Yao observables* of $\mathbf{V}$, written $\text{DY-obs}(\mathbf{V})$, are the values that an observer can obtain by splitting pairs into constituent values and using any private keys it possesses to decrypt ciphertexts it possesses. Formally, for a set of values $\mathbf{V}$, $\text{DY-obs}(\mathbf{V})$ is the least superset of $\mathbf{V}$, such that

- if $(v_1, v_2) \in \text{DY-obs}(\mathbf{V})$ then $v_1 \in \text{DY-obs}(\mathbf{V})$ and $v_2 \in \text{DY-obs}(\mathbf{V})$.
- if $\mathsf{sk}\langle i\rangle \in \text{DY-obs}(\mathbf{V})$ and $\boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle} \in \text{DY-obs}(\mathbf{V})$ then $v \in \text{DY-obs}(\mathbf{V})$

Second, given a value v, and a set of private keys $\mathbf{S}$, the *value pattern* of v with respect to $\mathbf{S}$, denoted $pat(v, \mathbf{S})$, is a template that describes which values an attacker that possesses $\mathbf{S}$ cannot distinguish from v. In particular, any ciphertext $\boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle}$ that appears in v for which the attacker does *not* possess the corresponding private key $\mathsf{sk}\langle i\rangle$ is replaced by $\boxed{\cdot}^{\mathsf{r}\langle j\rangle}$, indexed by the randomness used to generate the ciphertext. As mentioned before, we use this index to allow the attacker to distinguish identical ciphertexts from distinct ciphertexts, even when the attacker cannot decrypt the ciphertexts. The definition of $pat(v, \mathbf{S})$ is given in Figure 4.

We lift value patterns to lists of values: given a list of values

$$pat(\mathsf{NaV}, \mathbf{S}) = \mathsf{NaV} \qquad pat(n, \mathbf{S}) = n$$

$$pat(\mathsf{sk}\langle i\rangle, \mathbf{S}) = \mathsf{sk}\langle i\rangle \qquad pat(\mathsf{pk}\langle i\rangle, \mathbf{S}) = \mathsf{pk}\langle i\rangle$$

$$pat(\boxdot^{\mathsf{r}\langle j\rangle}, \mathbf{S}) = \boxdot^{\mathsf{r}\langle j\rangle}$$

$$pat((v_1, v_2), \mathbf{S}) = (pat(v_1, \mathbf{S}), pat(v_2, \mathbf{S}))$$

$$pat(\boxed{v}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle}, \mathbf{S}) = \begin{cases} \boxed{pat(v, \mathbf{S})}^{\mathsf{r}\langle j\rangle}_{\mathsf{pk}\langle i\rangle} & \text{if } \mathsf{sk}\langle i\rangle \in \mathbf{S} \\ \boxdot^{\mathsf{r}\langle j\rangle} & \text{if } \mathsf{sk}\langle i\rangle \notin \mathbf{S} \end{cases}$$

Fig. 4. Value pattern $pat(v, \mathbf{S})$

$\vec{v} = v_1, \ldots, v_n$, define the *list pattern*, denoted as $pattern(\vec{v})$, to be the list of values $pat(v_1, \mathbf{S}), \ldots, pat(v_n, \mathbf{S})$, where $\mathbf{S}$ is the set of observable private keys within $\vec{v}$, and is defined as $\mathbf{S} \triangleq \text{DY-obs}(\{v_i \mid 1 \leq i \leq n\}) \cap \{\mathsf{sk}\langle i\rangle \mid i \in \mathbb{N}\}$.

Using the above concepts, we can define *equivalence up to formal randomness*: two lists of values are equivalent up to formal randomness iff the list patterns of the two lists are identical, up to permutations on the indices of key pairs and random strings. Intuitively, two lists of values are equivalent up to formal randomness if a Dolev-Yao attacker is unable to distinguish the two lists.

Definition 4 (Equivalence up to formal randomness)**.** *Given two lists of values $\vec{v} = v_1, v_2, \ldots, v_n$ and $\vec{v'} = v'_1, v'_2, \ldots, v'_n$, say that these lists are equivalent up to formal randomness, written $\vec{v} \overset{\mathsf{rnd}}{=} \vec{v'}$, when there exist two permutations $\phi, \psi : \mathbb{N} \to \mathbb{N}$ such that $pattern(\vec{v})$ is identical to $pattern(\vec{v'})$ when public and private key indices are permuted according to ϕ and random string indices are permuted according to ψ, i.e.,*

$$pattern(\vec{v}) = pattern(\vec{v'})[{}^{\mathsf{pk}\langle\phi(1)\rangle}/_{\mathsf{pk}\langle 1\rangle}, {}^{\mathsf{pk}\langle\phi(2)\rangle}/_{\mathsf{pk}\langle 2\rangle}, \ldots, {}^{\mathsf{sk}\langle\phi(1)\rangle}/_{\mathsf{sk}\langle 1\rangle}, {}^{\mathsf{sk}\langle\phi(2)\rangle}/_{\mathsf{sk}\langle 2\rangle}, \ldots, {}^{\mathsf{r}\langle\psi(1)\rangle}/_{\mathsf{r}\langle 1\rangle}, {}^{\mathsf{r}\langle\psi(2)\rangle}/_{\mathsf{r}\langle 2\rangle}, \ldots]$$

Note that the definition uses permutations ϕ and ψ on keys and random strings respectively, implying that the specific keys and ciphertexts are not significant, but rather how these keys and ciphertexts are used.

Example. The following program generates two key pairs and then depending on whether $x > 0$ encrypts the integer 0 using one of the two keys before outputting the resulting ciphertext.

```
1 newkeypair(pk, sk, H); newkeypair(pk', sk', H);
2 if x > 0 then v := encrypt(pk, 0) else v := encrypt(pk', 0)
3 output(L, v);
```

Consider two executions of this program: one where $x = 5$ and the other where $x = 0$. The first execution results in the output $u_1 = \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 0\rangle}$; the second results in the output $u_2 = \boxed{0}^{\mathsf{r}\langle 1\rangle}_{\mathsf{pk}\langle 1\rangle}$. The corresponding list patterns of u_1 and u_2 are both $\boxdot^{\mathsf{r}\langle 0\rangle}$ and $\boxdot^{\mathsf{r}\langle 1\rangle}$. Using the permutation that maps 0 to 1 for both keys and random strings, we can derive that $u_1 \overset{\mathsf{rnd}}{=} u_2$. This matches our intuition: in both executions the attacker observes a single ciphertext that is unable to decrypt.

Example. The following program generates a key pair, encrypts the integer 0 and outputs the ciphertext, and then either outputs the same ciphertext, or another encryption of 0, depending on expression $x > 0$.

```
1 newkeypair(pk, sk, H); u := encrypt(pk, 0); output(L, u);
2 if x > 0 then v := u else v := encrypt(pk, 0);
3 output(L, v)
```

Consider two executions of this program that differ in the expression $x > 0$. The execution where $x > 0$ outputs values $u_1 = \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 0\rangle}, v_1 = \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 0\rangle}$. The other execution, where $x \leq 0$, outputs values $u_2 = \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 0\rangle}, v_2 = \boxed{0}^{\mathsf{r}\langle 1\rangle}_{\mathsf{pk}\langle 0\rangle}$. The list pattern of these outputs are $\boxdot^{\mathsf{r}\langle 0\rangle}, \boxdot^{\mathsf{r}\langle 0\rangle}$ and $\boxdot^{\mathsf{r}\langle 0\rangle}, \boxdot^{\mathsf{r}\langle 1\rangle}$ respectively. There are no permutations for which these two lists of values can satisfy Definition 4. Again, this matches our intuition: an attacker can distinguish an execution where the same ciphertext is output twice from an execution where two different ciphertexts are output, even though the attacker is unable to decrypt the ciphertexts.

Example. The following program is similar to the first example, but outputs the two secret keys before doing any encryptions.

```
1 newkeypair(pk, sk, ℓ); newkeypair(pk', sk', ℓ);
2 output(ℓ, (sk, sk'));
3 if x > 0 then u := encrypt(pk, 0) else u := encrypt(pk', 0)
4 output(ℓ, u);
```

Again, consider two executions that differ in the value of $x > 0$. One of these executions results in output sequence $(\mathsf{sk}\langle 0\rangle, \mathsf{sk}\langle 1\rangle), \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 0\rangle}$, while the other one results in $(\mathsf{sk}\langle 0\rangle, \mathsf{sk}\langle 1\rangle), \boxed{0}^{\mathsf{r}\langle 0\rangle}_{\mathsf{pk}\langle 1\rangle}$. The list patterns for these executions are not equivalent, because the attacker is able to use the private keys to decrypt the ciphertext, and thus distinguish them.

B. Cryptographic erasure

Using the notion of equivalence up to formal randomness, we can now extend our definitions of attacker knowledge and security to the extended language.

Definition 5 (Cryptographic attacker knowledge)**.** *Given program c, security level ℓ, and a subtrace t_{obs} observed by the attacker, define* cryptographic attacker knowledge *for subtrace t_{obs} at level ℓ as*

$$k^{\mathsf{rnd}}_{\ell}(c, t_{\mathsf{obs}}) \triangleq \{m \mid \langle c, m, \mathsf{U}_{\mathsf{init}}, \mathsf{Y}_{\mathsf{init}}, \varepsilon\rangle \longrightarrow^* \langle c_1, m_1, \mathsf{U}_1, \mathsf{Y}_1, t_1\rangle \longrightarrow^* \langle c_2, m_2, \mathsf{U}_2, \mathsf{Y}_2, t_1 \cdot t'\rangle \wedge \lfloor t'\rfloor_{\ell} \overset{\mathsf{rnd}}{=} \lfloor t_{\mathsf{obs}}\rfloor_{\ell}\}$$

The structure and intuition of this definition is similar to Definition 1. The only difference is the use of equivalence up to formal randomness between trace projections.

Definition 6 (Security with cryptography)**.** *Program c is* secure *for initial erasure environment γ if for all memories m and executions $\langle c, m, \mathsf{U}_{\mathsf{init}}, \mathsf{Y}_{\mathsf{init}}, \varepsilon\rangle_{\gamma} \longrightarrow^* \langle c_i, m_i, \mathsf{U}_i, \mathsf{Y}_i, t_i\rangle_{\gamma_i} \longrightarrow^* \langle c_n, m_n, \mathsf{U}_n, \mathsf{Y}_n, t_i \cdot t_{\mathsf{obs}}\rangle_{\gamma_n}$ where $i \leq n$, it holds that for all levels ℓ, we have $k^{\mathsf{rnd}}_{\ell}(c, t_{\mathsf{obs}}) \supseteq \bigcap_{i \leq j \leq n} \mathsf{ind}_{\ell}(m, \gamma_j)$*

Again, the structure and intuition of this definition is similar to Definition 3 from Section II-C, differing only in the use of cryptographic attacker knowledge.

Example. The following variant of our Snapchat-like example demonstrates how encrypting the compressed message before writing it to store makes the example secure.

```
1 newkeypair(pk, sk, L ^delete↗ H);
2 display := msg; output(L, display);
3 compressed := encrypt(pk, gzip(msg));
4 init store to compressed;
5 display := 0; msg := 0; sk := 0; set(delete);
6 output(L, display); output(L, sk);
7 read store into tmp; output(L, tmp);
```

This program first generates a key pair that can be used to encrypt information up to policy $\mathtt{L}^{delete}\!\nearrow\!\mathtt{H}$, and encrypts the compressed content of msg, storing the ciphertext in $compressed$. At line 5, variables $display$, msg and sk are overwritten and the condition $delete$ is set (requiring erasure of the initial value of variable msg and any information derived from that value). This program is secure: an attacker that starts observing program execution after $delete$ is set will learn nothing about the initial value of msg, even if the program subsequently outputs the contents of all memory locations, including the ciphertext value of $store$ (line 7). Intuitively, this program is secure because both the initial value of msg and the private key used to encrypt the information derived from that value were erased prior to $delete$ being set, even though an encryption of the compressed message is still accessible. If the assignment to sk in line 5 was omitted (i.e., the private key was not overwritten), the program would be insecure: after $delete$ is set, an observer could learn the private key and thus decrypt the ciphertext and learn the initial value of msg.

IV. Enforcement

In this section, we develop an enforcement mechanism for secure erasure based on a flow-sensitive security type system. A straightforward extension of our mechanism combines our type system with a run-time enforcement mechanism for a language with a heap. This extension is inspired by the work of Chong and Myers [13] and is described in an accompanying technical report [9].

In flow sensitive type systems, the type of a variable may change during program execution if the variable is updated. For example, in program $x := y$, where, initially, variable x is typed as H and variable y is typed as L, the type of x may be updated to L after the assignment.

The intuition behind our mechanism is the following:

1) The type system prevents information flows through variables or locations that enforce a weaker policy than the information requires. For example, the type system allows information with erasure policy $p\ ^{cnd}\!\nearrow\!q$ to flow to variables with policy $p\ ^{cnd}\!\nearrow\!\top$, but prevents flows to variables with policy p, since those variables would not be erased to q when the condition cnd is set. We formalize this relationship between security policies using a *relabeling judgment* (Section IV-A).

2) A command $\mathsf{set}(cnd)$ can only be typed when the typing environment contains no variables with security policies that contain erasure conditions cnd. In combination with (1), this requirement ensures that a condition cnd can be set only when no program variable contains information that needs to be erased by the time cnd is set (Section IV-B).

3) Encryption of sensitive information transfers the burden of preventing information flows from the data to the cryptographic keys used to encrypt it. As long as the keys are protected in accordance with the original policy on the information, the policy on the ciphertext can be relaxed (Section IV-C).

$$\textsf{r-lhs}\ \frac{p \le p' \qquad q \le p'}{p\ ^{c}\!\nearrow\!q \le p'} \qquad \textsf{r-rhs}\ \frac{p' \le p \qquad p' \le q}{p' \le p\ ^{c}\!\nearrow\!q}$$

$$\textsf{r-lab}\ \frac{\ell \sqsubseteq \ell'}{\ell \le \ell'} \qquad \textsf{r-samecond}\ \frac{p_1 \le p_2 \qquad q_1 \le q_2}{p_1\ ^{c}\!\nearrow\!q_1 \le p_2\ ^{c}\!\nearrow\!q_2}$$

Fig. 5. Relabeling judgment on policies

A. Relabeling judgment

Our first step for constructing an enforcement mechanism, is to define a *relabeling* judgment that we can use to compare erasure policies. We say that policy p can be *relabeled* to q, written $p \le q$, when enforcement of policy q on some information implies enforcement of policy p on the same information.

Figure 5 defines the relabeling judgment. Erasure policies consisting of labels are compared using the ordering of the underlying security lattice, as per rule r-lab. For compound policies, the rule r-lhs specifies that a policy $p\ ^{cnd}\!\nearrow\!q$ may be relabeled to policy p' if both p and q can be relabeled to p'. The condition variable cnd can be ignored because policy p' is restrictive enough to satisfy the requirements of both policies p and q, regardless of the condition variable. Rule r-rhs is similar. Finally, rule r-samecond defines relabeling between two policies with the same condition variable; $p_1\ ^{c}\!\nearrow\!q_1$ can be relabeled to $p_2\ ^{c}\!\nearrow\!q_2$ provided $p_1 \le p_2$ and $q_1 \le q_2$).

The relabeling judgment is sound with respect to the semantic interpretation of erasure policies. That is, strengthening the policy required on information in accordance with the relabeling order strictly decreases the number of initial memories an attacker can distinguish.

B. Flow-sensitive enforcement of erasure

Using the relabeling judgment, we introduce a security type system that enforces information erasure. Our type system has the following types:

$$b ::= \mathsf{int} \mid \mathsf{cond} \mid \mathsf{pubkey}_p \mid \mathsf{privkey}_p \mid \mathsf{enc}_p\ b \mid (b, b)$$
$$\tau ::= b\ p$$

Base types, denoted by b, range over integers, erasure conditions, public and private keys, encrypted values, and tuples (b, b). Types for public and private keys and encryptions

$$\frac{}{\Gamma \vdash \ell \ \mathsf{ok_{pol}}} \qquad \frac{p_1 \leq p_2 \qquad \Gamma(cnd) = \mathsf{cond}\ q \qquad q \leq p_i \qquad \Gamma \vdash p_i \ \mathsf{ok_{pol}} \qquad (i = 1, 2)}{\Gamma \vdash p_1 \overset{cnd}{\nearrow} p_2 \ \mathsf{ok_{pol}}}$$

Fig. 6. Well-formed erasure policies

are annotated with a key policy p. For keys the policy bounds the level of information that may be encrypted with the key pair. For encrypted values, the policy is same as the policy of the key that was used to produce the ciphertext. General types are denoted by τ and consist of a base type b and an associated policy p.

Typing environments and well-formed policies. Let Γ be a typing environment that maps standard variables and write-once variables to general types: $\Gamma : \mathbf{Vars} \cup \underline{\mathbf{Vars}} \to \tau$. We use $\dot{x}$ to range over the domain of Γ, while we use $\underline{x}$ and x when we refer specifically to write-once and standard variables respectively. We require that typing environments are *well-formed*. Defining well-formed typing environments requires first defining well-formed policies.

Figure 6 defines when a policy p is well-formed in Γ, denoted $\Gamma \vdash p \ \mathsf{ok_{pol}}$. Erasure policies that are simple security levels ℓ are always well formed. There are two key restrictions for a well-formed policy $p_1 \overset{cnd}{\nearrow} p_2$. First, the policy of the condition variable $\Gamma(cnd)$ must be no more restrictive than either p_1 or p_2. That is, information about whether cnd has been set is allowed to flow to both p_1 and p_2. This prevents covert information flows via condition variables. Second, policy p_2 must be as restrictive as p_1. Thus the policy is indeed an erasure policy rather than a declassification or an arbitrary relabeling.

We lift well-formedness from policies to typing environments. An environment Γ is well-formed, written $\vdash \Gamma \ \mathsf{ok_{env}}$, if and only if all policies it contains are well-formed (cf. Definition 7 in Appendix).

Flow-sensitive type system. The typing judgment for expressions has the form $\Gamma \vdash e : b\ p$. The intuition and inference rules for this judgment are the standard ones for security type systems, and we omit them for brevity.

The typing judgment for commands is flow sensitive, and has the form $\Gamma, pc \vdash c : \Gamma'$, where Γ is the typing environment immediately before the execution of command c, program counter policy pc is an erasure policy that is an upper bound on the information that may have influenced the decision to execute command c, and Γ' is the updated typing environment that holds immediately after command c terminates. Figure 7 presents selected typing rules for the non-cryptographic commands. In these rules, we use a straightforward extension of relabeling to types and typing environments. We require that all typing environments are well-formed, but omit the corresponding premises $\vdash \Gamma \ \mathsf{ok_{env}}$ from Figure 7 for brevity. The structure of the rules follows the flow-sensitive security type system of Hunt and Sands [26]. The interested reader can find the remaining rules of the type system in the Appendix.

Rule t-assign updates the erasure policy for variable x with

t-assign
$$\frac{\Gamma \vdash e : b\ p \qquad \mathsf{cond\text{-}free}(b) \qquad \{p, pc\} \leq q}{\Gamma, pc \vdash x := e : \Gamma[x \mapsto b\ q]}$$

t-init
$$\frac{\Gamma \vdash e : b\ p \qquad \Gamma(\underline{x}) = b\ q \qquad \{pc, p\} \leq q}{\Gamma, pc \vdash \mathsf{init}\ \underline{x}\ \mathsf{to}\ e : \Gamma}$$

t-read
$$\frac{\Gamma(\underline{x}) = b\ q \qquad \{pc, q\} \leq p}{\Gamma, pc \vdash \mathsf{read}\ \underline{x}\ \mathsf{into}\ y : \Gamma[y \mapsto b\ p]}$$

t-set
$$\frac{x \notin \mathsf{erasureconds}(\Gamma) \cup \mathsf{erasureconds}(pc) \qquad \Gamma(x) = \mathsf{cond}\ p \qquad pc \leq p}{\Gamma, pc \vdash \mathsf{set}(x) : \Gamma}$$

t-output
$$\frac{\Gamma \vdash e : b\ p \qquad \{p, pc\} \leq q \qquad \mathsf{cur}(q) \sqsubseteq \ell}{\Gamma, pc \vdash \mathsf{output}(\ell, e) : \Gamma}$$

Fig. 7. Selected typing rules for basic commands

policy q, where q is an upper bound of both the policy of expression e and the current program counter policy pc. This accounts for both explicit information flows from the computed expression and implicit information flows due to control flow Premise $\mathsf{cond\text{-}free}(b)$ ensures that the type of the assigned variable is not cond or a tuple that contains a cond. Thus only a $\mathsf{set}(x)$ command can mutate a condition variable.

The typing rules for write-once variables, t-init and t-read, are similar to the typing rule for assignment and expressions. Crucially, however, the types of write-once variables are flow-insensitive since information stored in them cannot be erased. The type system does not enforce that write-once variables are written to at most once (and thus the type system does not enforce progress). However, this property is not required to prove that the type system enforces security.

Rule t-set requires that when typing command $\mathsf{set}(x)$ neither variables in the current typing environment nor the program counter policy pc contain erasure policies with x as the erasure condition variable. This effectively ensures that any information that needs to be erased when the condition is set has already been erased by the program. Furthermore, it implies that information with an erasure policy that is eventually set cannot be stored in a write-once variable, since the type of write-once variables is fixed, and so the environment will always contain that condition variable. These restrictions are imposed by the requirement that $x \notin \mathsf{erasureconds}(\Gamma) \cup \mathsf{erasureconds}(pc)$. The type system also requires that $pc \leq p$ where $\Gamma(x) = \mathsf{cond}\ p$, which ensures that the decision to set the condition variable does not reveal sensitive information.

Operator $\mathsf{erasureconds}(arg_1, arg_2, \dots)$ returns all condition variables that appear in the erasure policies of its arguments (cf. Appendix). The erasure conditions of a variable with type $b\ p$

include condition variables within private key types appearing in b, ensuring that keys which could be used to decrypt sensitive information are also erased. Condition variables in public keys are not included because they do not protect sensitive information and do not need to be erased.

Rule t-output emphasizes the temporal nature of erasure policies: outputting a value on channel ℓ is permitted if the currently enforced security level of the value's policy is allowed to flow to ℓ. For example, if the policy $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$ is enforced on a value, then the currently enforced security level of the policy is L, and thus the value is allowed to be output to a channel with level L.

Example. Under initial typing environment Γ such that $\Gamma(cnd) = \mathsf{cond}\ \mathtt{L}$ and $\Gamma(x) = \mathsf{int}\ \mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$, and initial program counter policy $pc = \mathtt{L}$, the program

$$\mathsf{output}(\mathtt{L}, x); x := 0; \mathsf{set}(cnd); \mathsf{output}(\mathtt{L}, x)$$

is well-typed. While x initially contains information that requires erasure, it has been overwritten with public information before the command $\mathsf{set}(cnd)$. Thus the program type checks. Both output statements are secure since the first occurs while information with policy $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$ is allowed to be observed on channel L and the second outputs only public information.

Example. Under the same initial typing environment, the following program

$$\mathsf{if}\ x > 0\ \mathsf{then}\ \mathsf{set}(cnd)\ \mathsf{else}\ \mathsf{skip}$$

is rejected by the type system because information in x is not erased by the time cnd is set.

Example. Consider the program

$$\mathsf{output}(\mathtt{L}, x); \mathsf{init}\ \underline{file}\ \mathsf{to}\ x; x := 0;$$
$$\mathsf{set}(cnd); \mathsf{read}\ \underline{file}\ \mathsf{into}\ y; \mathsf{output}(\mathtt{L}, y)$$

under an initial program counter level $pc = \mathtt{L}$ and initial typing environment Γ where $\Gamma(cnd) = \mathsf{cond}\ \mathtt{L}$, $\Gamma(x) = \mathsf{int}\ \mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$, and $\Gamma(\underline{file}) = \mathsf{int}\ \mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$. This program is similar to the first example, but stores the sensitive information in x to $\underline{file}$ that cannot be erased. Right before $\mathsf{set}(cnd)$ maps $\underline{file}$ to $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$. Thus $\mathsf{set}(cnd)$ does not type check. Alternatively, if the initial environment mapped $\underline{file}$ to H, $\mathsf{set}(cnd)$ would type check, but $\mathsf{output}(\mathtt{L}, y)$ would not.

C. Typing cryptography

Figure 8 introduces the additional typing rules for cryptographic operations. These rules control the flow of information in the program and ensure that encrypted values and keys can be safely used to enforce information erasure.

Values of type $\mathsf{pubkey}_{p_k}\ p$ and $\mathsf{privkey}_{p_k}\ q$ are public and private keys, respectively, where p_k is an upper bound on the information that those keys can encrypt and decrypt. Thus in the rule for encryption (t-pub-enc), the information to be encrypted must be bounded above by p_k, and in the rule for decryption (t-pub-dec), the result of decryption must be bounded below by p_k.

In rule t-pub-newkey, the policy q enforced on a private key must be at least as restrictive as p_k and the decision to generate

t-pub-newkey

$$\frac{pc \le p \qquad pc \le p_k \qquad p_k \le q \qquad \Gamma' = \Gamma[x \mapsto \mathsf{pubkey}_{p_k}\ p, y \mapsto \mathsf{privkey}_{p_k}\ q]}{\Gamma, pc \vdash \mathsf{newkeypair}(x, y, p_k) :\ \Gamma'}$$

t-pub-enc

$$\frac{\Gamma \vdash e_1 : \mathsf{pubkey}_{p_k}\ p \qquad \Gamma \vdash e_2 : b\ p'' \qquad p'' \le p_k \qquad \{pc, p\} \le p' \qquad \Gamma' = \Gamma[x \mapsto \mathsf{enc}_{p_k}\ b\ p']}{\Gamma, pc \vdash x := \mathsf{encrypt}(e_1, e_2) : \Gamma'}$$

t-pub-dec

$$\frac{\Gamma \vdash e_1 : \mathsf{privkey}_{p_k}\ p \qquad \Gamma \vdash e_2 : \mathsf{enc}_{p_k}\ b\ q \qquad \{q, p_k, p, pc\} \le q' \qquad \Gamma' = \Gamma[x \mapsto b\ q']}{\Gamma, pc \vdash x := \mathsf{decrypt}(e_1, e_2) : \Gamma'}$$

Fig. 8. Typing for cryptography

a new key pair (p_c) must be bounded below by p_k . Intuitively, if secret information is encrypted, the private key that can decrypt it should also be treated as secret information; if the private key were public information, then anyone could use the private key to decrypt the ciphertext and obtain secret information. Crucially, this restriction also ensures that if encrypted information needs to be erased, it suffices to erase the private key. That is, if a keypair can be used to encrypt information that must be erased, then the policy enforced on the private key must be at least as restrictive as the erasure policy, and the private key must be erased if needed. A single key-pair may be used to protect multiple data items, so long as the bound of the key pair is at least as restrictive as the erasure policy to enforce on each sensitive datum.

In rule t-pub-enc, the ciphertext that results from encrypting a value with policy p'' using a public key with type $\mathsf{pubkey}_{p_k}\ p$ has policy p', and p' does not need to be bounded below by either p'' or p_k. That is, the result of encrypting a secret value is a non-secret ciphertext. Note, however, that both the decision to encrypt and the choice of which public key to use may reveal information, and so p' must be bounded below by both p and pc. Similarly, in rule t-pub-dec the result of decrypting a ciphertext must be bound below by the decision to decrypt, the choice of which private key to use, and the choice of which ciphertext to decrypt.

Example. Recall the secure variant of the Snapchat-like example from Section III. Consider an initial typing environment $\Gamma(delete) = \mathsf{cond}\ \mathtt{L}$, $\Gamma(msg) = \mathsf{int}\ \mathtt{L} \overset{delete}{\nearrow} \mathtt{H}$, and $pc = \mathtt{L}$. Also assume that the type of $gzip(msg)$ is the same as the type of msg, i.e, the type captures the implicit flow from msg to the result of the expression. This program is well typed, since the ciphertext $compressed$ may have policy L and thus $\underline{store}$ does not need to be erased before the condition is set. Moreover, as we discuss in Section III, the secret key sk is erased. Thus this program is also semantically secure.

Example. Consider the same program but with the command $sk := 0$ removed. This program is no longer secure because an observer can use the secret key to decrypt the second output,

learning information about msg after the condition is set. This program is rejected by the type system because sk is not treated in accordance with the key policy $\mathtt{L} \overset{delete}{\nearrow} \mathtt{H}$, which would require it to be erased before the command $\mathsf{set}(delete)$.

D. Assurance

We now formulate our main soundness result. First, observe that our type system is termination insensitive [8] and thus cannot enforce Definition 6. Instead, we state soundness with respect to a weaker, termination-insensitive variant of Definition 6. Existing techniques that either combine type-based enforcement with termination analysis [36] or further restrict the expressiveness of programs to eliminate termination channels [39, 44, 48] can mitigate this limitation.

As a technical device in our definition of termination-insensitive security, we introduce $\mathsf{Conv}(c)$, the set of converging memories.

$$\mathsf{Conv}(c) \triangleq \{m \mid \langle c, m, \mathsf{U}_{\mathsf{init}}, \varepsilon\rangle \longrightarrow^* \langle \mathsf{stop}, m', \mathsf{U}', t'\rangle\}$$

In the statement of our security theorem below, we use the set of converging memories to exclude memories that would cause the program to diverge or get stuck.

Theorem 1 (Termination-insensitive security)**.** *Consider program c such that $\Gamma, pc \vdash c : \Gamma'$, Then for all $m \in \mathsf{Conv}(c)$ and all runs such that $\langle c, m, \mathsf{U}_{\mathsf{init}}, \mathsf{Y}_{\mathsf{init}}, \varepsilon\rangle_\gamma \longrightarrow^* \langle c_i, m_i, \mathsf{U}_i, \mathsf{Y}_i, t_i\rangle_{\gamma_i} \longrightarrow^* \langle c_n, m_n, \mathsf{U}_n, \mathsf{Y}_n, t_i \cdot t_{\mathsf{obs}}\rangle_{\gamma_n}$, where $i \leq n$, it holds that for all levels ℓ:*

$$k_\ell^{\mathsf{rnd}}(c, t_{\mathsf{obs}}) \supseteq \mathsf{Conv}(c) \cap \bigcap_{i \leq j \leq n} \mathsf{ind}_\ell(m, \gamma_j)$$

Standard proof techniques to show that a security type system enforces a noninterference-like property do not immediately work. There are two complicating factors. First, we need to prove a noninterference-like property for security policies ordered by the relabeling relation. However, channel outputs violate this ordering (since, e.g., information labeled $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$ can be output to channel L, even though $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H} \not\leq \mathtt{L}$). This requires proving a property stronger than noninterference, where, for example, even though two L-equivalent executions may start with different values at policy $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$, once condition cnd is set, the two executions are thereafter $\mathtt{L} \overset{cnd}{\nearrow} \mathtt{H}$-equivalent. Second, our erasure policies are dynamic policies [4], and we must show that the type system is a sound and accurate static approximation of these policies. This requires powerful invariants, including that branches and loops modify the type of a variable only if the branch or loop might modify the variable's value. The accompanying technical report [9] includes the proof of Theorem 1 for our model extended with a heap enforcement mechanism. Theorem 1 as stated in this section is a corollary.

V. Implementation

We have implemented an extended version of the enforcement mechanism from Section IV in the tool KEYRASE. KEYRASE combines static analysis and source-to-source translation to enforce erasure policies in Java programs. To effectively support erasure in the presence of objects, the tool combines a flow-sensitive analysis to ensure erasure of information in local variables with a run-time mechanism to erase data in the heap. The run-time mechanism allows information with erasure policies to flow into the heap without requiring precise static reasoning. The extension of the language to model a heap and run-time erasure mechanism is presented in the technical report [9].

KEYRASE uses the Accrue Interprocedural Java Analysis Framework [14] (or, simply, "Accrue"). Accrue is itself built as a compiler extension to Polyglot [38], an extensible compiler framework for Java. KEYRASE is implemented in approximately 3,000 lines of Java code, but is built of on top of an interprocedural constraint-based information-flow analysis, which consists of approximately 11,000 lines of Java code. KEYRASE uses contraint-based type inference to require fewer type annotations.

Language extensions. KEYRASE extends Java with a distinguished type `Condition`. Conditions may appear only as static fields or local variables, must be declared `final`, and cannot be used in expressions or passed as arguments. This ensures that conditions used in erasure policies can always be statically identified. A program sets a condition with its `set` method.

Programmers may annotate field declarations and expressions with a security policy for the contents of the field or the result of the expression. Erasure policies that appear in annotations can refer to any condition variable in scope. KEYRASE also has annotations for declassification. While our formal model does not include declassification, we conjecture that it can be accounted for by weakening the indistinguishability relation in the style of Askarov and Sabelfeld [5].

Security analysis. When invoked, KEYRASE attempts to infer security policies for every field, local variable, and expression in a program and reports whether those policies are consistent with information flow constraints and annotations. The constraints track explicit and implicit information flows through Java language features such as objects, fields, static and dynamic method dispatch, and exceptions. The constraints do not track information flow via termination, timing, or other covert channels. Like the enforcement mechanism described in Section IV, our analysis is flow sensitive for local variables and expressions but flow insensitive for heap locations. We do not support concurrency.

In addition to constraints that ensure standard information-flow restrictions, KEYRASE uses several constraints to enforce the novel requirements of our type system: (1) Whenever a condition may be set (either by calling `set` directly, or by calling a method or constructor that may set a condition), we require that the condition does not appear in the security policies inferred for either the program counter or local variables in scope at that program point. This is analogous to the requirements of typing rule t-set. (2) At calls to encryption and decryption functions, we constrain the policies associated with public and private keys according to typing rules t-pub-enc and t-pub-dec. (3) By default, fields are constrained to simple security level policies unless they are explicitly annotated with an erasure policy. Furthermore, to make sure that fields with

erasure policies can be erased by reassignment, they may not be marked `final`.

We rely on programmer-provided signatures to identify methods for key generation, encryption and decryption. The programmer may also provide signatures describing the information-flow behavior of library code. When neither code nor a signature is available, KEYRASE (unsoundly) assumes that information may flow from the receiver and arguments of a method call to the return value of the method and to all fields of the receiver.

Translation. KEYRASE implements a source-to-source translation to enforce erasure in the heap. The translation injects code into the constructors or static initializers of classes that contain fields that may require erasure. This code registers a listener with the condition specified in the erasure policy. When the condition is set, it notifies the listeners, which overwrite the fields with default values.

VI. CASE STUDY

We used KEYRASE to check the security of examples in this paper. We also implemented a simple Internet Relay Chat (IRC) client that keeps a compressed and encrypted log of messages on disk. The client comprises 316 non-comment lines of code (excluding libraries) and can connect to a server, join a chat channel, send and receive messages, and view previous messages from the log.

The client also has a `\clear` command that erases all information about previous chat messages from the system. This security requirement is expressed as an erasure policy on received messages: $\texttt{L}^{\,\mathsf{clear}\nearrow\top}$. This policy requires information to be erased when the condition clear is set, which happens when the `\clear` command is issued. Afterwards, the client can resume logging new messages.

To reflect that data on disk cannot be reliably deleted, the signatures for methods that write to the file system accept only information with policy `L`. Thus, the client cannot log messages to the file system directly. Instead, it compresses and encrypts the log in screen-sized blocks before writing it to disk. KEYRASE ensures that the keys used are also protected with policy $\texttt{L}^{\,\mathsf{clear}\nearrow\top}$ and thus cannot be written to the file system. Information about received messages also flows to the component that maintains a short buffer of messages used to redraw the screen. This buffer must also be erased when the clear condition is set. Of course, the terminal window itself may contain sensitive information. The client erases this trusted channel by sending a clear screen command.

Other information flows in the program reveal sensitive information indirectly: for example, writing even encrypted information to the file system reveals information about the existence of messages, if not their contents. KEYRASE detects this leak because the client attempts to write to the file system while the program counter is tainted. Since the information revealed is minimal, we added a declassification annotation to allow it rather than modify the program to always write a fixed-size log entry.

KEYRASE also detected leaks caused by insecure programming. For example, in Java, many methods that operate on sensitive data can throw exceptions that leak information by preventing subsequent outputs that would otherwise be secure. Avoiding these leaks requires careful handling of exceptions, which is difficult without tool support.

In total, the IRC client required eight annotations: two to declare the types of cryptographic keys, two to declare the erasure policies of object fields, and four declassification annotations. One is the declassification of whether a new log entry has been added, described above. The remaining three declassify (1) the number of previous log entries, (2) output to the terminal window, and (3) whether a message was a "ping" (which generates multiple log entries).

Our experience with KEYRASE confirms that correctly identifying sensitive data that should be erased is challenging—most of the information flows we found were not immediately obvious. This reinforces the importance of cryptographic information erasure rather than data deletion in applications that use untrusted storage.

VII. RELATED WORK

Information erasure. Chong and Myers introduce language-based information erasure [12] and define erasure policies via a denotational semantics with an attacker that has direct access to memory. Our semantics for erasure policies is operational. In addition we use a precise and extensible knowledge-based attacker model, which allows for an extension to cryptographically-masked values [7].

Additional work has furthered the semantic definition of information erasure. Del Tedesco et al. [16] provide a semantic framework for expressive erasure policies. Their semantic condition, like ours, is based on attacker knowledge. Their policies can describe richer erasure policies than ours, including erasure policies that depend on the data to be erased. In contrast to our work, though, they do not provide an enforcement mechanism. Beyond the sequential programs that our work focuses on, Jiang et al. [28] consider information erasure and release in multi-threaded programs, using a bisimulation-based semantic condition.

There are several language-based mechanisms for information erasure. However, unlike our work, none support cryptographic enforcement. Hunt and Sands [27] give the first such non-cryptographic mechanism, but support only lexically delimited erasure conditions. This restriction enables simple and elegant enforcement via a flow-sensitive security-type system. Hansen and Probst [24] enforce erasure policies in Java Card bytecode, with a single erasure condition set at the end of execution. Nanevski et al. [37] specify and enforce erasure policies using Relational Hoare Type Theory. Del Tedesco et al. [15] design a Python library for erasure but ignore implicit flows. Chong and Myers [13] enforce dynamic erasure conditions using an expensive run-time mechanism that requires local variables that may need erasure to be lifted to the heap. Our tool, KEYRASE combines static enforcement of erasure for local variables with a run-time enforcement mechanism

similar to that of Chong and Myers [13] for the heap. Thus KEYRASE mitigates the scalability issues of an exclusively run-time mechanism.

Van Delft et al. [46] describe a general framework for enforcing dynamic policies like information erasure. Similar to our approach, they separate the enforcement of the policy into a flow-sensitive dependency analysis and a static approximation of the policy. In fact, proving the correctness of the static approximation is the most challenging aspect of our proofs.

Deletion without guaranteed erasure. Perlman proposes a file system that provides "assured deletion" [40], where data is stored encrypted and trusted stores (called *ephemerizers*) are responsible for managing cryptographic keys, including erasing them when needed. Tang et al. [45] implement a service on Amazon S3 that realizes these ideas. Dunn et al. [18] use similar techniques to implement *ephemeral channels* that enable assured deletion of communication between a process and peripheral devices, and Bauer and Priyantha [10] propose similar techniques to securely delete files and backups. Several encrypted file systems use this approach to securely delete files (e.g., [33, 34, 41, 50]). Our work extends this line of work. It (i) provides firm foundations for assured deletion and (ii) extends it to information erasure.

Cryptography and information-flow control. We build on prior work on information-flow security for languages with cryptographic primitives. Volpano and Smith [47, 49] consider the secure use of one-way functions in programs. Abadi [1] introduces information-flow control type systems for a language with symmetric cryptographic operators. Abadi and Blanchet [2] consider asymmetric cryptography operators, and Fournet et al. [19] consider operators homomorphic encryption. Laud [29] considers encryption operators, but assumes that keys can be statically distinguished [30]. Askarov et al. [7] introduce (and enforce) a semantic security condition that can reason about keys that can be dynamically (but not statically) distinguished. These prior works use a symbolic model of cryptography [17], which Laud proves sound for a computational model of cryptography [31]. Laud and Vene [32] present a type system that is sound for a computational model of cryptography. Our work is inspired by these results. In particular, Askarov et al. [7] and Laud [31] provide the starting point for our exploration of the use of cryptography for information erasure — a direction beyond the goals of previous work on cryptography-based information security.

VIII. CONCLUSION

This work combines cryptographic data deletion with information-flow control. The result is a novel semantic security condition for cryptographic information erasure, a flow-sensitive security type system that enforces it, and an implementation (and case study) that demonstrates the practicality of the approach. This work improves the practicality of language-based information erasure and provides stronger guarantees than existing cryptographic data deletion.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant No. 1054172 and by the Air Force Research Laboratory. We would like to thank Anitha Gollamudi for her helpful feedback.

REFERENCES

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, Sept. 1999.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

[3] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, pages 79–94, 2010.

[4] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium*, pages 308–322, 2012.

[5] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221, 2007.

[6] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the IEEE Computer Security Foundations Symposium*, July 2009.

[7] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. In *Proceedings of the 13th International Static Analysis Symposium*, Lecture Notes in Computer Science, Aug. 2006.

[8] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, 2008.

[9] A. Askarov, S. Moore, C. Dimoulas, and S. Chong. Cryptographic enforcement of language-based information erasure. Technical Report TR-01-15, Harvard School of Engineering and Applied Sciences, 2015. URL ftp://ftp.deas.harvard.edu/techreports/tr-01-15.pdf.

[10] S. Bauer and N. B. Priyantha. Secure data deletion for Linux file systems. In *Proceedings of the 10th Conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[11] D. Boneh and R. J. Lipton. A revocable backup system. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, page 9, Berkeley, CA, USA, 1996. USENIX Association.

[12] S. Chong and A. C. Myers. Language-based information erasure. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[13] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111, June 2008.

[14] S. Chong, A. Johnson, S. Moore, and O. Arden. Accrue Interprocedural Java Analysis Framework, 2013. http://people.seas.harvard.edu/~chong/accrue.html.

[15] F. Del Tedesco, A. Russo, and D. Sands. Implementing erasure policies using taint analysis. In *Proceedings of the Nordic Conference in Secure IT Systems*, 2010.

[16] F. Del Tedesco, S. Hunt, and D. Sands. A semantic hierarchy for erasure policies. In *Seventh International Conference on Information Systems Security*, 2011.

[17] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[18] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel. Eternal sunshine of the

spotless machine: Protecting privacy with ephemeral channels. In *10th USENIX Symposium on Operating Systems Design and Implementation*, pages 61–75. USENIX, 2012.

[19] C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 351–360, 2011.

[20] S. L. Garfinkel and A. Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17–27, Jan. 2003.

[21] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *The Sixth USENIX Security Symposium Proceedings*, pages 77–90, 1996.

[22] P. Gutmann. Data remanence in semiconductor devices. In *The Tenth USENIX Security Symposium Proceedings*, pages 39–54, 2001.

[23] J. Guynn. Privacy watchdog EPIC files complaint against Snapchat with FTC. Los Angeles Times, May 17 2013.

[24] R. R. Hansen and C. W. Probst. Non-interference and erasure policies for Java Card bytecode. In *Proceedings of the 6th International Workshop on Issues in the Theory of Security*, Mar. 2006.

[25] K. Hill. Snapchats don't disappear: Forensics firm has pulled dozens of supposedly-deleted photos from Android phones. Forbes, May 9 2013.

[26] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 79–90, Jan. 2006.

[27] S. Hunt and D. Sands. Just forget it—the semantics and enforcement of information erasure. In *Proceedings of the 17th European Symposium on Programming*, pages 239–253, 2008.

[28] L. Jiang, L. Ping, and X. Pan. Handling information release and erasure in multi-threaded programs. In *Computational Intelligence and Security: International Conference*, pages 824–828, Dec. 2007.

[29] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 77–91, 2001.

[30] P. Laud. Handling encryption in analyses for secure information flow. In *Proceedings of the 12th European Symposium on Programming*, pages 159–173, 2001.

[31] P. Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*, Jan. 2008.

[32] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proceedings of the 15th international conference on Fundamentals of Computation Theory*, pages 365–377, 2005.

[33] B. Lee, K. Son, D. Won, and S. Kim. Secure data deletion for usb flash memory. *Journal of Information Science and Engineering*, 27:933–952, 2011.

[34] J. Lee, S. Yi, J. Heo, H. Park, S. Y. Shin, and Y. Cho. An efficient secure deletion scheme for flash file systems. *Journal of Information Science and Engineering*, 26:27–38, 2010.

[35] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *Computer Security–ESORICS 2013*, pages 57–74. 2013.

[36] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 881–893, Oct. 2012.

[37] A. Nanevski, A. Banerjee, and D. Garg. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems*, 35 (2), 2013.

[38] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *12th International Conference on Compiler Construction*, pages 138–152, 2003.

[39] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201, June 2006.

[40] R. Perlman. File system design with assured delete. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[41] Z. N. J. Peterson, R. Burns, J. Herring, A. Stubblefield, and A. D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies*, page 11, Berkeley, CA, USA, 2005. USENIX Association.

[42] J. Reardon, D. Basin, and S. Capkun. Sok: Secure data deletion. *2013 IEEE Symposium on Security and Privacy*, pages 301–315, 2013.

[43] K. Satvat, M. Forshaw, F. Hao, and E. Toreini. On the privacy of private browsing - a forensic approach. *Journal of Information Security and Applications*, 19(1):88–100, 2014.

[44] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[45] Y. Tang, P. Lee, J. Lui, and R. Perlman. Secure overlay cloud storage with access control and assured deletion. *Dependable and Secure Computing, IEEE Transactions on*, 9(6):903–916, Nov. 2012.

[46] B. van Delft, S. Hunt, and D. Sands. Very static enforcement of dynamic policies. In R. Focardi and A. Myers, editors, *Principles of Security and Trust*, volume 9036 of *Lecture Notes in Computer Science*, pages 32–52. Springer Berlin Heidelberg, 2015.

[47] D. Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations*, pages 246–254, 2000.

[48] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE Computer Security Foundations Workshop*, pages 156–168, 1997.

[49] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Conference Record of the Twenty-Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 268–276, Jan. 2000.

[50] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Cucs-021-98, Columbia University, 1998.

APPENDIX

A. Enforcement of information erasure

This section presents the complete set of rules of the type system of Section IV for the basic commands of the language of Section II, i.e., our language model without cryptographic primitives. Before delving into the rules, we introduce a few auxiliary definitions that for brevity we describe only informally in Section IV.

1) Well-formedness of environments: We lift well-formedness from policies to typing environments. An environment Γ is well-formed if and only if all erasure policies in Γ are well-formed. To formalize this definition, we introduce an auxiliary function $\mathsf{nestedpols}(\kappa, b)$ that returns the set of erasure policies appearing within a base type b.

The argument κ is a key type selector, and is a subset of the set of strings $\{\mathsf{privkey}, \mathsf{pubkey}\}$. Operator nestedpols is defined as follows:

$$\mathsf{nestedpols}(\kappa, \mathsf{privkey}_p) = p \text{ if } \mathsf{privkey} \in \kappa$$
$$\mathsf{nestedpols}(\kappa, \mathsf{pubkey}_p) = p \text{ if } \mathsf{pubkey} \in \kappa$$
$$\mathsf{nestedpols}(\kappa, (b_1, b_2)) = \mathsf{nestedpols}(\kappa, b_1) \cup \mathsf{nestedpols}(\kappa, b_2)$$
$$\mathsf{nestedpols}(\kappa, b) = \emptyset \text{ otherwise}$$

For a base type b we refer to $\mathsf{nestedpols}(\{\mathsf{privkey}\}, b)$ as the set of the critical policies within b—this set includes all nested policies except those appearing in public keys, and is used in a few places within the type system. With this operator, we can now define well-formedness of a typing environment.

Definition 7 (Well-formedness of typing environments)**.** *A typing environment* Γ *is* well formed*, written* $\vdash \Gamma\ \mathsf{ok}_{\mathsf{env}}$*, if and only if for all* $\dot{x}$ *such that* $\Gamma(\dot{x}) = b\ p$ *and all policies* q *such that* $q \in \{p\} \cup \mathsf{nestedpols}(\{\mathsf{privkey}, \mathsf{pubkey}\}, b)$*,* $\Gamma \vdash q\ \mathsf{ok}_{\mathsf{pol}}$*.*

2) The predicate $\mathsf{cond\text{-}free}(b)$*:* The predicate $\mathsf{cond\text{-}free}(b)$ succeeds when b is not cond or a tuple that contains a cond. The predicate is defined recursively as follows:

$$\mathsf{cond\text{-}free}(\mathsf{int}).$$
$$\mathsf{cond\text{-}free}(\mathsf{pubkey}_p).$$
$$\mathsf{cond\text{-}free}(\mathsf{privkey}_p).$$
$$\mathsf{cond\text{-}free}(\mathsf{enc}_p\ b') \text{ if } \mathsf{cond\text{-}free}(b').$$
$$\mathsf{cond\text{-}free}((b', b'')) \text{ if } \mathsf{cond\text{-}free}(b') \text{ and } \mathsf{cond\text{-}free}(b'').$$

3) The operator $\mathsf{erasureconds}(p)$*:* The operator $\mathsf{erasureconds}(arg_1, arg_2, \dots)$ returns all condition variables that appear in the erasure policies of its arguments:

$$\mathsf{erasureconds}(\ell) = \emptyset$$
$$\mathsf{erasureconds}(p_1 \overset{x}{\nearrow} p_2) = \{x\} \cup \mathsf{erasureconds}(p_1)$$
$$\mathsf{erasureconds}(p_1, p_2, \dots) = \mathsf{erasureconds}(p_1) \cup \mathsf{erasureconds}(p_2) \cup \dots$$
$$\mathsf{erasureconds}(\Gamma) = \bigcup_{\Gamma(x) = b\ p} \mathsf{erasureconds}(p) \cup \{\mathsf{erasureconds}(q) \mid q \in \mathsf{nestedpols}(\{\mathsf{privkey}\}, b) \wedge \neg(q \leq p)\}$$

4) The operator $\uplus$*:* The *constrained merge* operator $\Gamma_1 \uplus \Gamma_2$ returns an environment that combines Γ_1 and Γ_2:

$$\Gamma_1 \uplus \Gamma_2 = \lambda \dot{x}.\ \begin{cases} \Gamma_1(\dot{x}) & \text{if } \Gamma_1(\dot{x}) = \Gamma_2(\dot{x}) \\ \tau & \text{if } \Gamma_1(\dot{x}) \neq \Gamma_2(\dot{x}) \\ & \quad \text{where } \Gamma_i(\dot{x}) \leq \tau, \quad i = 1, 2 \end{cases}$$

5) The type system rules: With the above definitions in hand, we can now provide the inference rules of the flow-sensitive type system of Section IV for the basic commands of the language model of Section II:

t-skip
$$\frac{}{\Gamma, pc \vdash \mathsf{skip} : \Gamma}$$

t-assign
$$\frac{\Gamma \vdash e : b\ p \qquad \mathsf{cond\text{-}free}(b) \qquad \{p, pc\} \leq q}{\Gamma, pc \vdash x := e : \Gamma[x \mapsto b\ q]}$$

t-init
$$\frac{\Gamma \vdash e : b\ p \qquad \Gamma(\underline{x}) = b\ q \qquad \{pc, p\} \leq q}{\Gamma, pc \vdash \mathsf{init}\ \underline{x}\ \mathsf{to}\ e : \Gamma}$$

t-read
$$\frac{\Gamma(\underline{x}) = b\ q \qquad \{pc, q\} \leq p}{\Gamma, pc \vdash \mathsf{read}\ \underline{x}\ \mathsf{into}\ y : \Gamma[y \mapsto b\ p]}$$

t-set
$$\frac{x \notin \mathsf{erasureconds}(\Gamma) \cup \mathsf{erasureconds}(pc) \qquad \Gamma(x) = \mathsf{cond}\ p \qquad pc \leq p}{\Gamma, pc \vdash \mathsf{set}(x) : \Gamma}$$

t-output
$$\frac{\Gamma \vdash e : b\ p \qquad \{p, pc\} \leq q \qquad \mathsf{cur}(q) \sqsubseteq \ell}{\Gamma, pc \vdash \mathsf{output}(\ell, e) : \Gamma}$$

t-seq
$$\frac{\Gamma_0, pc \vdash c_1 : \Gamma_1 \qquad \Gamma_1, pc \vdash c_2 : \Gamma_2}{\Gamma_0, pc \vdash c_1; c_2 : \Gamma_2}$$

t-if
$$\frac{\Gamma \vdash e : \mathsf{int}\ p \qquad \{p, pc\} \leq pc' \qquad \Gamma, pc' \vdash c_1 : \Gamma_1 \qquad \Gamma, pc' \vdash c_2 : \Gamma_2}{\Gamma, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \Gamma_1 \uplus \Gamma_2}$$

t-while
$$\frac{\begin{array}{c}\Gamma_i \vdash e : \mathsf{int}\ p_i \\ \{p_i, pc\} \leq pc_i \qquad \Gamma_i, pc_i \vdash c : \Gamma_i' \quad (0 \leq i \leq n) \\ (\text{where } \Gamma_{i+1} = \Gamma_i' \uplus \Gamma_0, pc_i \leq pc_{i+1}, \\ \Gamma_{n+1} = \Gamma_n, pc_{n+1} = pc_n)\end{array}}{\Gamma_0, pc \vdash \mathsf{while}\ e\ \mathsf{do}\ c : \Gamma_n}$$

$$\Gamma_1 \uplus \Gamma_2 = \lambda \dot{x}.\ \begin{cases} \Gamma_1(\dot{x}) & \text{if } \Gamma_1(\dot{x}) = \Gamma_2(\dot{x}) \\ \tau & \text{if } \Gamma_1(\dot{x}) \neq \Gamma_2(\dot{x}) \\ & \quad \text{where } \Gamma_i(\dot{x}) \leq \tau, \quad i = 1, 2 \end{cases}$$