Match It or Die: Proving Integrity by Equality ⋆

Matteo Centenaro and Riccardo Focardi

Università Ca' Foscari Venezia,
{mcentena,focardi}@dsi.unive.it

Abstract. Cryptographic hash functions are commonly used as modification detection codes. The goal is to provide message integrity assurance by comparing the digest of the original message with the hash of what is thought to be the intended message. This paper generalizes this idea by applying it to general expressions instead of just digests: success of an equality test between a tainted data and a trusted one can be seen as a proof of high-integrity for the first item. Secure usage of hash functions is also studied with respect to the confidentiality of digests by extending secret-sensitive noninterference of Demange and Sands.

Keywords: Information Flow, Language-based Security, Security Type System, Hash Functions.

1 Introduction

A hash function takes as input an arbitrary message and 'summarizes' it into a *digest*. In cryptography, hash functions are commonly used as *modification detection codes* [9]: we are guaranteed of the integrity of a message M whenever the hash of M coincides with a previously computed digest of the original message. Moreover, hash functions are also commonly employed to protect data secrecy as done, e.g., in Unix password files. To provide both integrity and confidentiality, hash functions are required to respectively be *collision resistant* and *one-way* [9], meaning that it should be infeasible to exhibit two messages with the same digest and to find a message whose digest matches a given one.

A first example of everyday usage of hash functions is password-based authentication: a one-way hash of the user password is securely stored in the system and is compared with the hash of the password typed by the user at the login prompt, whenever the user wants to access her account. The following code is a simplified fragment of the Unix *su* utility used to let a system administrator perform privileged actions. The password file is modeled as an array `passwd[username]`.

⋆ Work partially supported by Miur'07 Project SOFT: "*Security Oriented Formal Techniques*"

```
trial = hash(t_pwd);
if (trial = passwd[root]) then
    << launch the administrator shell >>
```

The typed password `t_pwd` is given as input to the program thus we regard it as untrusted. In fact, from the program perspective there could be an enemy 'out there' trying to impersonate the legitimate administrator. The same holds for `trial`, which is computed from an untrusted value. Existing type systems for noninterference would consequently consider the guard of the if branch as tainted, or low-integrity, since its value is computed from untrusted data and possibly under the control of the enemy. For this reason, the code in the if-then branch would be required to never modify high-integrity values. Clearly the administrator shell can make any change to the system including, e.g., modifying user passwords, and this program would be consequently rejected.

One of the motivations for hashing passwords is to protect confidentiality. In fact, if the hash function is one-way, it is infeasible for an opponent to find a password whose hash matches the one stored in the password file. In practice, brute force dictionary attacks suggest that the password file should be nevertheless kept inaccessible to non administrators, as it is done, e.g., in the *shadow password* mechanism of Unix. However, if password entropy is 'high enough' it might be safe to let every user access the hashed passwords. Formally, this would correspond to assigning the array `passwd[]` a low-confidentiality security level. Consider now the following update of Alice's password to the new, high-confidentiality value `alice_pwd`:

```
passwd[alice] := hash(alice_pwd);
```

This assignment would be rejected by usual type systems for noninterference, as it *downgrades* the confidentiality level of the password.

Hash functions are also often used for integrity checks. We consider a software producer who wants to distribute an application on the Internet, using different mirrors in order to speedup the downloads. A common way to assure users downloading a binary file `my_blob.bin` from mirrors of its integrity, is to provide them with a trusted digest `swdigest` of the original program. The browser would then run a code similar to the following:

```
if (hash(myblob.bin) = swdigest) then
    trusted_blob.bin := my_blob.bin;
    << install trusted_blob >>
```

The idea is that the user will install the given binary only if its digest matches the one of the original program provided by the software company. In fact, if the hash function is collision resistant, it would be infeasible for an attacker to modify the downloaded program while preserving the digest. Once the check succeeds, `my_blob.bin` can be safely 'promoted' to high-integrity and installed into the system. This is modelled by assigning `my_blob.bin` to the high-integrity variable `trusted_blob.bin`. This is usually regarded as a direct integrity flaw and rejected by usual type systems for noninterference. Moreover, installing the application can be thought as writing into a high-integrity area of the file system and, as for the root shell above, would be forbidden in a branch with a low-integrity guard.

Our contribution. We have discussed how typical examples of programs that use cryptographic hash functions break standard notions of noninterference, even if they are intuitively secure. In this work, we study how to extend noninterference notions so that such kinds of program can be type checked and proved secure. We model hash functions symbolically: the hash of a value v is simply $\mathsf{h}(v)$. We do not assume any deconstructor allowing to recover v from $\mathsf{h}(v)$ thus modelling the fact h is one-way, and we also assume $\mathsf{h}(v) = \mathsf{h}(v')$ if and only if $v = v'$, modelling collision resistance. As is customary in symbolic settings, what has negligible probability in the computational world becomes here impossible.

We focus on what we informally call 'match-it-or-die' programs which, like the above examples, always perform integrity checks at the outer level and fail whenever the check is not passed. For these programs, the attacker is not interested in causing a failure, as no code would be executed in such a case. This enables us to type check programs that assign to high-integrity variables even in a low-integrity if branch, as in the Unix *su* example. We then observe that assignments such as `trusted_blob.bin := my_blob.bin` are safe under the check `hash(myblob.bin) = swdigest`. In fact, since `swdigest` is high-integrity, when its value matches the one of `hash(myblob.bin)`, we are guaranteed that `myblob.bin` has not been tampered with. This allows us to type check programs like the application downloading example.

Moreover, we investigate the confidentiality requirements for using hash functions to preserve data secrecy. We first observe that if the entropy of the hashed value is low an attacker might try to compute, by brute force, the hash of all the possible values until he finds a match. We thus select, as our starting point, a recent noninterference variant

called *secret-sensitive* noninterference [6] which distinguishes small and big secrets and allows us to treat their corresponding digests accordingly. If a secret is big, meaning that it is infeasible to guess its actual value, then the brute force attack above is also infeasible. We show that it is safe to downgrade the hash of a big secret, assuming some control over what secret is actually hashed. In fact, two hashes of the same big secret are always identical and the opponent might deduce some information by observing equality patterns of digests. This requires a nontrivial extension of the notion of memory equivalence so to suitably deal with such equality patterns.

Finally, we give a security type system to statically enforce that programs guarantee the proposed noninterference notions.

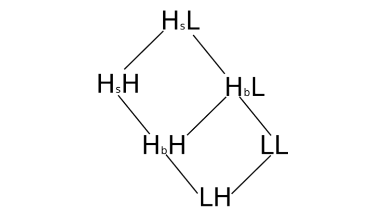
Structure of the paper. In Section 2 we give the background on secret-sensitive noninterference [6]; Section 3 extends the noninterference notions so to correctly deal with hash functions. Integrity check by equality is analyzed in Section 4. The security type system enforcing noninterference is given in Section 5 while Section 6 discusses related works. The paper closes with some final remarks and ideas for future work in Section 7.

2 Secret-sensitive Noninterference

Secret-sensitive noninterference (SSNI) [6], by Demange and Sands, is a variant of noninterference which distinguishes small, guessable secrets from big, unguessable ones. As discussed in the introduction, this distinction will be useful to discipline the downgrading of digests of secret values, as we will see in Section 3.

Size aware security lattice. Secrets are partitioned into big ($\mathsf{H_b}$) and small ($\mathsf{H_s}$) ones. Preorder $\sqsubseteq_C$ among confidentiality levels is defined as $\mathsf{L} \sqsubseteq_C \mathsf{H_b} \sqsubseteq_C \mathsf{H_s}$, meaning that public, low data can be regarded as secret and, as discussed above, small secrets need to be treated more carefully than big ones. We extend this size aware confidentiality lattice by composing it with the basic two-level integrity lattice in which $\mathsf{H} \sqsubseteq_I \mathsf{L}$. Notice that integrity levels are contravariant: low-integrity, tainted values have to be used more carefully than high-integrity, untainted ones. The product

Fig. 1 Size aware Lattice



of these two lattices is depicted in Figure 1. We will write $\ell = \ell_C \ell_I$ to range over the product lattice elements. The ordering between the new security levels ℓ is denoted by $\sqsubseteq$ and is defined as the componentwise application of $\sqsubseteq_C$ and $\sqsubseteq_I$.

Language. Programs under analysis are written in a standard imperative while-language. We assume a set of variables *Var* ranged over by x, a set of values *Val* ranged over by v and a set of arithmetic and boolean operations ranged over by op . The syntax for expressions e and commands c follows.

$$e ::= x \mid e_1 \;\mathsf{op}\; e_2$$

$$\mathsf{c} ::= \mathsf{skip} \mid x := e \mid \mathsf{if}\; e \;\mathsf{then}\; \mathsf{c}_1 \;\mathsf{else}\; \mathsf{c}_2 \mid \mathsf{while}\; e \;\mathsf{do}\; \mathsf{c} \mid \mathsf{c}_1; \mathsf{c}_2$$

Memories $\mathsf{M} : \mathit{Var} \rightarrow \mathit{Val}$ are finite maps from variables to values. We write $e \downarrow^{\mathsf{M}} v$ to note the atomic evaluation of expression e to value v in memory M. Command semantics is given in terms of a small-step transition between configurations $\langle \mathsf{M}, \mathsf{c} \rangle$. Transitions are labeled with an *event* $\alpha \in \mathit{Var} \cup \{\tau\}$ indicating that an assignment to variable α (or no assignment if α is τ) has happened. Command semantics rules are standard and are omitted here for lack of space; they can be found in the full version of the paper [4].

Observable behaviour. We assume to have a *security environment* Γ mapping variables to their security levels. Users at level ℓ may only read variables whose level is lower or equal than ℓ. Let $\mathsf{M}_{|\ell}$ be the projection of the memory M to level ℓ, i.e., memory M restricted to variables visible at level ℓ or below.

Definition 1 (Memories ℓ-equivalence). *M and M' are ℓ-equivalent, written $\mathsf{M} =_\ell \mathsf{M}'$, if $\mathsf{M}_{|\ell} = \mathsf{M}'_{|\ell}$.*

Intuitively, users at level ℓ will never be able to distinguish two ℓ-equivalent memories M and M'.

Similarly, users may only observe transitions assigning to variables at or below their level. Transition $\xrightarrow{\alpha}_\ell$ is defined as the least relation among configurations such that:

$$\frac{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{x} \langle \mathsf{M}', \mathsf{c}' \rangle \qquad \Gamma(x) \sqsubseteq \ell}{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{x}_\ell \langle \mathsf{M}', \mathsf{c}' \rangle}$$

$$\frac{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\alpha} \langle \mathsf{M}', \mathsf{c}' \rangle \qquad \alpha = \tau \text{ or } \alpha = x \text{ with } \Gamma(x) \not\sqsubseteq \ell}{\langle \mathsf{M}, \mathsf{c} \rangle \xrightarrow{\tau}_\ell \langle \mathsf{M}', \mathsf{c}' \rangle}$$

We write $\stackrel{\alpha}{\Rightarrow}_\ell$ to denote $\stackrel{\tau}{\rightarrow}_\ell^{*}\stackrel{\alpha}{\rightarrow}_\ell$, if $\alpha \neq \tau$, or $\stackrel{\tau}{\rightarrow}_\ell^{*}$ otherwise. Transitions $\stackrel{\tau}{\Rightarrow}_\ell$ are considered internal, silent reductions which are unobservable by anyone. Notice, instead, that for observable transitions $\stackrel{x}{\Rightarrow}_\ell$, the level of x is always at or below ℓ, i.e., $\Gamma(x) \sqsubseteq \ell$.

Secure programs. The main idea of SSNI [6] is that for unguessable secrets, brute force attacks will terminate only with negligible probability. Intuitively, this allows for adopting a termination insensitive equivalence notion when comparing program behaviour. Guessable secrets, instead, can be leaked by brute force using 'termination channels', and for those values it is necessary to distinguish between terminating and nonterminating executions.

A configuration $\langle \mathsf{M}, \mathsf{c}\rangle$ diverges for ℓ, written $\langle \mathsf{M}, \mathsf{c}\rangle \Uparrow_\ell$, if it will never perform any ℓ-observable transition $\stackrel{x}{\rightarrow}_\ell$. A termination insensitive ℓ-bisimulation requires that observable transitions of the first program are simulated by the second one, unless the latter diverges on ℓ meaning that it cannot execute any observable action at or below ℓ.

Definition 1 (**Termination insensitive ℓ-bisimulation**)
A symmetric relation $\mathcal{R}$ on configurations is a termination insensitive ℓ-bisimulation (ℓ-TIB) *if* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \mathcal{R} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ *implies* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *and whenever* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \stackrel{\alpha}{\rightarrow}_\ell \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ *then either*

- $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \stackrel{\alpha}{\Rightarrow}_\ell \langle \mathsf{M}'_2, \mathsf{c}'_2\rangle$ *and* $\langle \mathsf{M}'_1, \mathsf{c}'_1\rangle \mathcal{R} \langle \mathsf{M}'_2, \mathsf{c}'_2\rangle$ *or*
- $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \Uparrow_\ell$

Configurations $\langle \mathsf{M}_1, \mathsf{c}_1\rangle$, $\langle \mathsf{M}_2, \mathsf{c}_2\rangle$ *are* termination insensitive ℓ-bisimilar*, written* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \approx_\ell \langle \mathsf{M}_2, \mathsf{c}_2\rangle$*, if there exists a ℓ-TIB relating them.*

Termination sensitive bisimulation, instead, always requires observable actions to be simulated.

Definition 2 (**Termination sensitive ℓ-bisimulation**)
A symmetric relation $\mathcal{R}$ on configurations is a termination sensitive ℓ-bisimulation (ℓ-TSB) *if* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \mathcal{R} \langle \mathsf{M}_2, \mathsf{c}_2\rangle$ *implies* $\mathsf{M}_1 =_\ell \mathsf{M}_2$ *and whenever* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \stackrel{\alpha}{\rightarrow}_\ell \langle \mathsf{M}'_1, \mathsf{c}'_1\rangle$ *then* $\langle \mathsf{M}_2, \mathsf{c}_2\rangle \stackrel{\alpha}{\Rightarrow}_\ell \langle \mathsf{M}'_2, \mathsf{c}'_2\rangle$, $\langle \mathsf{M}'_1, \mathsf{c}'_1\rangle \mathcal{R} \langle \mathsf{M}'_2, \mathsf{c}'_2\rangle$. *Configurations* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle$ *and* $\langle \mathsf{M}_2, \mathsf{c}_2\rangle$ *are* termination sensitive ℓ-bisimilar*, written* $\langle \mathsf{M}_1, \mathsf{c}_1\rangle \simeq_\ell \langle \mathsf{M}_2, \mathsf{c}_2\rangle$*, if there exists a ℓ-TSB relating them.*

A secure program will preserve small secrets from being leaked via the termination channel while will be more liberal with respect to the big ones. This is achieved by requiring termination sensitive bisimilarity whenever the inspected memories are the same at level $\mathsf{H_bL}$, meaning they only

differ on small secrets. Notice that, as usual, the attacker is assumed to be at level LL.

Definition 3 (**Secret-sensitive NI**)
A command c *satisfies* secret-sensitive NI *if* $\forall\ \mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$ *it holds*

1. $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$ *and*
2. $\mathsf{M}_1 =_{\mathsf{H_bL}} \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c} \rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

3 Hash Functions and Secrecy

This section extends secret-sensitive NI to programs that use hash functions. As already observed, hash functions could be subject to brute force attacks, unless the hashed messages are big enough to make exhaustive search infeasible. The idea is to take advantage of the two distinct secret levels $\mathsf{H_b}$ and $\mathsf{H_s}$ to protect digests of small secrets and treat more liberally the digests of big secrets.

Hash expressions. The language of the previous section is augmented with a new hash expression whose semantics is defined in terms of a special constructor h. Formally, $\mathsf{hash}(e) \downarrow^{\mathsf{M}} \mathsf{h}(v)$ if $e \downarrow^{\mathsf{M}} v$ with $v \in \mathit{Val}$. We then partition Val into the sets of small and big values Val_s, Val_b, ranged over by v_s and v_b. We define the sets of small and big digests as $\mathit{Val}^d_\delta = \{\mathsf{h}(v) \mid v \in \mathit{Val}_\delta\}$, with $\delta \in \{s, b\}$. As discussed in the introduction, this simple modelling of hash functions is coherent with the assumption of being one-way (no deconstructor expressions) and collision resistant (digests of different values *never* collide).

Memories ℓ-equivalence. Lifting the notion of memory equivalence when dealing with digests requires to carefully handle equality patterns. In fact, in our symbolic model, equal digests will correspond to equal hashed messages.

Consider the program $x := \mathsf{hash}(y)$, where x is a public variable and y is a secret one. It must be considered secure only if y is a big secret variable, indeed leaking the digest of a small secret is equivalent to directly reveal the secret since an attacker could perform a brute force attack on the hash.

The equivalence notion between memories needs, however, to be relaxed in order to capture the fact that big secrets are, in practice, random unpredictable values. We illustrate considering again $x := \mathsf{hash}(y)$ and assuming y to be a big secret variable. We let $\mathsf{M}_1(x) = 0 = \mathsf{M}_2(x)$,

$\mathsf{M}_1(y) = v_b \neq v'_b = \mathsf{M}_2(y)$, then it holds $\mathsf{M}_1 =_{\mathsf{LL}} \mathsf{M}_2$. Executing the above code the resulting memories differ on the value stored in the public variable x: $\mathsf{M}'_1(x) = \mathsf{h}(v_b) \neq \mathsf{h}(v'_b) = \mathsf{M}'_2(x)$. It follows that $\mathsf{M}'_1 \neq_{\mathsf{LL}} \mathsf{M}'_2$ and the program does not respect noninterference so it would be rejected as insecure. However, v_b and v'_b are two big random numbers and we can never expect they are equal. Thus, the only opportunity for the attacker is to see if they correspond to other big values in the same memory. Requiring the equality of big secrets and digests across memories is, consequently, too strong.

This boils down to the idea of *patterns*, already employed in [5,7,8] for cryptographic primitives. We illustrate through an example. Consider program $z := \mathsf{hash}(x); w := \mathsf{hash}(y)$ where z and w are public variables and x and y are big secrets. Consider the following memories:

$$\begin{array}{l|l} \mathsf{M}_1 & \mathsf{M}_2 \\ \hline x : v_b & x : v_b \\ y : v'_b & y : v_b \\ z : 0 & z : 0 \\ w : 0 & w : 0 \end{array} \qquad (1)$$

executing the above code would make public two different digests in M_1 and the very same digests in M_2. The attacker is able to learn that the first memory stores two different secrets values while the second does not. In summary, we do not require the equality of big secrets and digests across memory but only that the equality patterns are the same.

As the last example shows, in order to safely downgrade digests of big secrets we need to control how big secrets are stored in the memories. We do this by projecting out from memories big secret values which are either stored in big secret variables or hashed and observable from ℓ. This is done by the following function r, taking as parameters the value v and the level ℓ_v of a variable.

$$\mathsf{r}_\ell(v, \ell_v) = \begin{cases} v \ \text{ if } v \in \mathit{Val}_b \text{ and } \ell_v = \mathsf{H_b}\ell_I \\ v' \text{ if } v = \mathsf{h}(v'), v' \in \mathit{Val}_b \text{ and } \ell_v \sqsubseteq \ell \\ 0 \ \text{ otherwise} \end{cases}$$

A *big secret projection* $\mathsf{r}_\ell(\mathsf{M})$ is defined as $\mathsf{r}_\ell(\mathsf{M})(x) = \mathsf{r}_\ell(\mathsf{M}(x), \Gamma(x))$, for all $x \in \mathit{Dom}(\mathsf{M})$. Two memories will be *comparable* if their big secret projections can be matched by renaming big values , i.e., if two big values are the same in one projection then it will also be the case that they are equal in the other one.

Definition 4 (Comparable memories)
Two memories M_1 *and* M_2 *are* ℓ-comparable*, noted* $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$*, if there exists a bijection* $\mu : Val_b \to Val_b$ *such that* $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$*.*

Example 1. The two above memories (1) are not comparable if observed at level LL, i.e., $\mathsf{M}_1 \not\bowtie_{\mathsf{LL}} \mathsf{M}_2$. In fact, $\mathsf{r}_\ell(\mathsf{M}_1)(x) = v_b \neq v'_b = \mathsf{r}_\ell(\mathsf{M}_1)(y)$ while $\mathsf{r}_\ell(\mathsf{M}_2)(x) = v_b = v_b = \mathsf{r}_\ell(\mathsf{M}_2)(y)$. Thus there exists no bijection μ such that $\mathsf{r}_\ell(\mathsf{M}_1) = \mathsf{r}_\ell(\mathsf{M}_2)\mu$, since μ cannot map v_b to both v_b and v'_b.

Two memories are ℓ-equivalent if they are ℓ-comparable and their observable big digests expose the same equality patterns. A *digest substitution* ρ is a bijection on digests of big values: $\rho : \; Val_b^d \to Val_b^d$.

Definition 5 (**Memory ℓ-equivalence with hash functions**)
Two memories, M_1 *and* M_2*, are* ℓ-equivalent*, written* $\mathsf{M}_1 =_\ell^h \mathsf{M}_2$*, if* $\mathsf{M}_1 \bowtie_\ell \mathsf{M}_2$ *and there exists a digest substitution* ρ *such that* $\mathsf{M}_{1|\ell} = \mathsf{M}_{2|\ell}\ \rho$*.*

Secure programs. The bisimulation definitions given in the previous section are left unchanged except for the relation used to compare memories which is now $=_\ell^h$ in place of $=_\ell$. Secret-sensitive NI is thus rephrased as follows.

Definition 6 (**Secret-sensitive NI with hash functions**)
A command c *satisfies* secret-sensitive NI *if* $\forall\ \mathsf{M}_1 =_{\mathsf{LL}}^h \mathsf{M}_2$ *it holds*

1. $\langle \mathsf{M}_1, \mathsf{c}\rangle \approx_{\mathsf{LL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$ *and*
2. $\mathsf{M}_1 =_{\mathsf{H_bL}}^h \mathsf{M}_2$ *implies* $\langle \mathsf{M}_1, \mathsf{c}\rangle \simeq_{\mathsf{H_bL}} \langle \mathsf{M}_2, \mathsf{c}\rangle$*.*

4 Proving Integrity by Equality

In a recent work [5], we prove integrity of low-integrity data using Message Authentication Codes (MACs). The secrecy of the MAC key ensures the integrity of the exchanged data: once the MAC is recomputed and checked, we are guaranteed that no one has manipulated the received data. This work generalizes the idea by applying it to the simpler case of an equality test with respect to a high-integrity value: if the test is successful we are guaranteed that the compared, low-integrity, value has not been tampered with.

Integrity can be checked via noninterference by placing the observer at level $\mathsf{H_sH}$. This amounts to quantifying over all the values in low-integrity variables and observing any interference they possibly cause on high-integrity variables.

Definition 7 (**Integrity NI**)
A program c *satisfies* integrity NI *if for all* $\mathsf{M}_1, \mathsf{M}_2$ *such that* $\mathsf{M}_1 =_{\mathsf{H_sH}} \mathsf{M}_2$ *it holds* $\langle \mathsf{M}_1, \mathsf{c} \rangle \approx_{\mathsf{H_sH}} \langle \mathsf{M}_2, \mathsf{c} \rangle$.

Consider the program `if (x = y) then` c_1 `else` c_2 where x is a low-integrity variable and y is a high-integrity one. If either c_1 or c_2 modifies high-integrity variables this program is rejected. If it were not, an opponent manipulating the low-integrity variable x might force the program to execute one of the two branches and gain control on the fact high-integrity variables are updated via c_1 or c_2.

Consider now the case of the simplified *su* utility discussed in the introduction. Similarly to what we have seen above, an attacker might insert a wrong administrator password making the check fail. However, the program is in what we have called match-it-or-die form: the else branch is empty and nothing is executed after the if-then command. In the definition we have given, we obtain that the program diverges and the termination insensitive notion of Integrity NI would consider the program secure.

A special case of integrity test is the one which involves the comparison between the on-the-fly hash of a low-integrity message and a trusted variable. Upon success, integrity of the untrusted data will be proved and it will be possible to assign it to a high-integrity variable. Consider the following program where y and z are trusted variables while x is a tainted one.

```
if (hash(x) = y) then
   z := x;
```

As in the software distribution example of the introduction the assignment is safe, since it will be executed only if the contents of the variable x has been checked to be high-integrity by comparing its digest with the high-integrity digest y.

5 Security Type System

This section presents a security type system to statically analyze programs that use hash functions and derive data integrity by equality tests.

The proposed solution is based on the type system by Demange and Sands [6]. We only report typing rules for expressions and integrity check commands. All the remaining rules are as in [6].

We distinguish among four different types of values: small, big and their respective digests. Value types VT are S (small), B (big), $\mathsf{S}^{\#}$ (hash

of a small) and $\mathsf{B}^{\#}$ (hash of a big) and are ranged over by vt. These value types are populated by the respective values:

$$\text{(v-small)}\ \frac{v \in \mathit{Val}_s}{\vdash v : \mathsf{S}}\ \text{(v-big)}\ \frac{v \in \mathit{Val}_b}{\vdash v : \mathsf{B}}\ \text{(v-hashs)}\ \frac{v \in \mathit{Val}_{ds}}{\vdash v : \mathsf{S}^{\#}}\ \text{(v-hashb)}\ \frac{v \in \mathit{Val}_{db}}{\vdash v : \mathsf{B}^{\#}}$$

Security types are of the form $\tau = \lambda\ell$, where $\lambda \in \{\mathsf{P}, \mathsf{D}\}$ distinguish between plain values and digests, while ℓ is the associated security level. A *security type environment* Δ is a mapping from variable to their security types. Given $\tau = \lambda\ell$ the two functions T and L give respectively its variable type and security level, i.e., $\mathsf{T}(\tau) = \lambda$ and $\mathsf{L}(\tau) = \ell$.

A subtype relation is defined over security types, it is meant to preserve λ, as we do not want to mix plain values with digests. Moreover plain big secrets do not appear in the relation meaning that they can only be picked from Val_b. Subtyping $\leq$ is the least relation such that $\tau_1 \leq \tau_2$ if $\mathsf{T}(\tau_1) = \mathsf{T}(\tau_2) = \mathsf{D}$ and $\mathsf{L}(\tau_1) \sqsubseteq \mathsf{L}(\tau_2)$ or $\mathsf{T}(\tau_1) = \mathsf{T}(\tau_2) = \mathsf{P}$, $\mathsf{L}(\tau_i) \neq \mathsf{H_b}\ell_I$ and $\mathsf{L}(\tau_1) \sqsubseteq \mathsf{L}(\tau_2)$.

To prove that the type system enforces the security properties stated above it must be that plain big secret variables really store big values. To guarantee that small values are never assigned to big variables a conservative approach will be taken: every expression which involves an operator and returns a plain value lifts the confidentiality level of its result to $\mathsf{H_s}$, whenever it would be $\mathsf{H_b}$. The following function on security levels performs this upgrade:

$$\ell^{\sqcup} = \begin{cases} \mathsf{H_s}\ell_I & \text{if } \ell = \mathsf{H_b}\ell_I \\ \ell & \text{otherwise} \end{cases}$$

A variable x respects its security type $\tau = \lambda\ell$ with respect to a memory M if $\lambda = \mathsf{P}$ and $\vdash \mathsf{M}(x) : \mathsf{S}$ or $\vdash \mathsf{M}(x) : \mathsf{B}$ and similarly if $\lambda = \mathsf{D}$ then $\vdash \mathsf{M}(x) : \mathsf{S}^{\#}$ or $\vdash \mathsf{M}(x) : \mathsf{B}^{\#}$. A memory will be said to be well-formed if it respects the type of its variables, more precisely the expected properties are:

1. All variable respects their security types
2. Public variables do not store plain big values
3. Plain big secret variables only store big values.

From now on it will be supposed that all the memories are well-formed and indeed it can be proved that memory well-formedness is preserved by typed programs.

Expression typing rules are depicted in Table 1. Rules (var) and (sub) are standard. Rule (eq) types the equality test of two expressions requiring that they type the same τ and judging the (boolean) result as a plain small

Table 1 Security Type System - The Most Significant Rules

Expressions

$$\text{(var)}\ \frac{\Delta(x) = \tau}{\Delta \vdash x : \tau} \qquad \text{(sub)}\ \frac{\Delta \vdash e : \tau' \quad \tau' \leq \tau}{\Delta \vdash e : \tau}$$

$$\text{(eq)}\ \frac{\Delta \vdash e : \tau \quad \Delta \vdash e_2 : \tau \quad \mathsf{L}(\tau) = \ell}{\Delta \vdash e_1 = e_2 : \mathsf{P}\ell^{\sqcup}} \qquad \text{(op)}\ \frac{\Delta \vdash e_1 : \mathsf{P}\ell \quad \Delta \vdash e_2 : \mathsf{P}\ell}{\Delta \vdash e_1\ \mathsf{op}\ e_2 : \mathsf{P}\ell^{\sqcup}}$$

$$\text{(hash-b)}\ \frac{\Delta \vdash x : \mathsf{PH_b}\ell_I}{\Delta \vdash \mathsf{hash}(x) : \mathsf{DL}\ell_I} \qquad \text{(hash-s)}\ \frac{\Delta \vdash x : \mathsf{P}\ell}{\Delta \vdash \mathsf{hash}(x) : \mathsf{D}\ell}$$

Integrity Test Commands

$$\text{(int-test)}\ \frac{\begin{array}{c}\Delta \vdash x : \tau \quad \Delta \vdash y : \tau' \quad \mathsf{T}(\tau) = \mathsf{T}(\tau') \\ \mathsf{L}(\tau) = \ell_C\mathsf{L} \quad \mathsf{L}(\tau') = \ell_C\mathsf{H} \qquad \ell_C \sqsubseteq_C \mathsf{H_b} \quad \Delta \vdash \mathsf{c} : (\ell_C\mathsf{H}, t, f)\end{array}}{\Delta \vdash \mathsf{if}\ x = y\ \mathsf{then\ c\ else\ FAIL} : (\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)}$$

$$\text{(int-hash)}\ \frac{\begin{array}{c}\Delta \vdash x : \mathsf{P}\ell_C\mathsf{L} \quad \Delta \vdash y : \mathsf{D}\ell_C\mathsf{H} \\ \Delta(z) = \mathsf{P}\ell_C\mathsf{H} \qquad \ell_C \sqsubseteq_C \mathsf{H_b} \quad \Delta \vdash \mathsf{c} : (\ell_C\mathsf{H}, t, f)\end{array}}{\Delta \vdash \mathsf{if\ hash}(x) = y\ \mathsf{then}\ z := x; \mathsf{c\ else\ FAIL} : (\ell_C\mathsf{H}, t \sqcup \ell_C\mathsf{L}, \uparrow)}$$

value. Rule (op) let any operator to be applied only to plain expressions, since in our symbolic model of the hash function no operation is defined on digests except for the equality test. These two rules use the $\ell^{\sqcup}$ function to promote the confidentiality level of their result to $\mathsf{H_s}$ whenever necessary, as already discussed above.

Hashes are typed either by (hash-b) or (hash-s). Rule (hash-b) performs a controlled declassification, the idea is that since the message is a plain big value its secrecy is not broken by releasing its digest. Indeed it can be proved that this does not break noninterference. The latter typing rule does nothing special and just preserve the security level of its argument.

The type system has to enforce a termination insensitive noninterference for big secrets and termination sensitive for small ones. This latter requirement can be achieved by some strong limitations on the while loops [13] or by accounting for the termination effect of a command [3,11]. Demange and Sands type system [6] is built upon the work of Boudol and Castellani [3]. For lack of space, we only illustrate the two new rules added to the SSNI original type system [6]. The remaining type system can be found in the full version of the paper [4].

A command type is a triple (w, t, f) where w and t are security levels and f is a termination flag ranging over $\downarrow$ and $\uparrow$ which respectively note

that the command always terminates or that it could not terminate. A program is considered to be always terminating if it does not contain any while loop. The two flags are ordered as $\downarrow \sqsubseteq \uparrow$. A type judgement of the form $\Delta \vdash \mathsf{c} : (w, t, f)$ means that c does not assign to variables whose security level is lower than w (w is the writing effect of c), observing the termination of c gives information on variables at most at level t (t is the termination effect of c) and the termination behaviour is described by f.

Rules (int-test) and (int-hash) are new contributions of this work and implement the integrity verification tests discussed in Section 4. The former one let a trusted computation happens if the integrity of a tainted variable is proved by an equality test with an untainted one. The latter is pretty similar but is specific for the hash case and asserts that the intended original message, stored in the untrusted variable used to compute the on-the-fly digest, can be assigned to a trusted variable, whenever the check succeeds.

The else branch in both cases must be the special command FAIL. It is a silent diverging while loop of the form while true do skip. Requiring that each integrity test command executes such a program in case its guard condition is not satisfied assure that all the typed programs are in the 'match-it-or-die' form. In fact, upon failure no observable actions will be ever executed which is equivalent to say that, from an attacker point of view, no code would be run.

The confidentiality level of the variables involved in (int-test) and (int-hash) guard is constrained to be at most $\mathsf{H_b}$ to avoid brute force attacks to small secrets. The command in the if branch (c) is typed with a writing effect which has an high-integrity level, thus letting it to have write clearance to high-integrity variables, and the same confidentiality level as the two expressions compared in the guard. The termination effect of the overall command is constrained by the one of c and by $\ell_C \mathsf{L}$ since the test contains variables which are at most at that security level. Note that the two integrity tests would be potentially non terminating due to the FAIL branch.

Results. The proposed type system enforces the security properties given in Section 3 and Section 4: If a program type checks then it is both secret-sensitive and integrity noninterferent.

Theorem 1 (**SSNI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies Secret-sensitive NI.*

Theorem 2 (**Integrity NI by typing**)
If $\Delta \vdash \mathsf{c} : (w, t, f)$ *then* c *satisfies integrity NI.*

Proofs are omitted here for lack of space but can be found in the full version of the paper [4]. In the Appendix the simplified *su* utility and the software distribution examples given in the introduction, are shown to type check. A program which let a user update its password is also presented and typed.

6 Related Works

This section discusses related work in the literature.

Hash functions. A secure usage of hash function in the setting of information flow security has been already explored by Volpano in [12]. There are, however, many difference with respect to our work. First, Volpano does not account for data integrity and, consequently, integrity checks, which is one of the major contributions of our work. On the other side, we limit our study to a symbolic treatment of hash functions, distinguishing between two different kind of secrets, while Volpano aims at a computational result.

Secrecy. The use of patterns to define a memory equivalence notion suitable to be used to compare digests, originates from the work of Abadi and Rogaway [2] and Abadi and Jürjens [1] whose aim was to establish a link between the formal and computational treatment of encryption. The same idea has recently been applied in the information flow security [7,8] to prove that randomized cyphertexts could be leaked without breaking noninterference. Deterministic encryption has been modeled in a symbolic setting for information flow by Centenaro, Focardi, Luccio and Steel [5] extending the idea of pattern. That work anyway does not account for hash functions. The distinction between big and small secrets and the two different bisimilarity notions which have to be applied to protect them is completely inspired by Demange and Sands [6].

Integrity checks. A recent work on the type checking of PIN processing security APIs [5] already implements an integrity test by means of Message Authentication Codes (MACs). It might appear that this was possible thanks to the usage of a secret key in the computation of the MAC. Instead, this work shed new light on the fundamental reasoning underlying such integrity checks, which are generalized to a less restrictive context and also applied to the special case of the hash function.

7 Conclusions

We have studied the security of programs that use hash functions in the setting of information flow security. We have shown how to prove data integrity via equality tests between a low and a high-integrity variable.

We have extended secret-sensitive noninterference to guarantee that leaks via the hash operator could not occur: the intuition is that the digest of a big enough secret s would not be subject to a brute force attack and so releasing it to the public will not break the confidentiality of s.

A classical noninterference property has been instead used to check that secure programs do not taint high-integrity data. Equality tests to enforce data integrity have been introduced: this idea is a generalization of the integrity proof performed using Message Authentication Codes (MACs) in [5]. The equality of a tainted variable with a trusted one is regarded as an evidence of the fact that the value stored in the untrusted variable is indeed untainted. This kind of integrity proof is widely adopted in real applications and this work gives the tools to reason about its security.

Future Work. Hash functions could be used in commitment protocols. Suppose Anna challenges Bruno to solve a problem and claims she has solved it. To prove her statement Anna takes the answer and appends it to a random secret *nonce*, she then sends the hash of such message to Bruno. When the challenge finishes or when Bruno gives up, Anna has to reveal him the secret nonce thus he can check that the correct answer was sent in the first step of the process.

Formally studying this scenario in an information flow setting would be challenging. Some form of declassification would be allowed since at certain point in time the secret nonce has to be released. When the random will be downgraded then the digest could not be thought to protect Anna's answer anymore. Analyzing the security of this problem will require an interaction of a declassification mechanism, suitable to reason about the *when* dimension of downgrading [10], with the solution presented here for the secure usage of hash functions.

To guarantee that small values are never assigned to big variables we have taken the very conservative approach of forbidding expressions to return a big secret. In practice, this might be relaxed by adding some data flow analysis in order to track values derived from big secrets. For example, the xor of two different big secrets might be considered a big secret, but the xor of two equal big secrets is 0. We intend to investigate this issue more in detail in the next future.

One of the anonymous reviewer has let us notice a strong similarity between our notion of memory equivalence, based on patterns, and the notion of *static equivalence* in process calculi. Big secrets resemble the notion of bound names which can be α-converted preserving equality patterns. We leave as a future work the intriguing comparison between the two formal notions.

Acknowledgements. We would like to thank the anonymous reviewers for their very helpful comments and suggestions.

References

1. Martín Abadi and Jan Jürjens. Formal eavesdropping and its computational interpretation. In *Proceedings of the 4th International Conference on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *LNCS*, pages 82–94, Tohoku University, Sendai, Japan, October 29-31 2001. Springer.
2. Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *JCRYPTOL: Journal of Cryptology*, 15(2):103–127, 2002.
3. Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *LNCS*, pages 382–395, Crete, Greece, July 2001. Springer.
4. Matteo Centenaro and Riccardo Focardi. Match it or die: Proving integrity by equality. `http://www.dsi.unive.it/~mcentena/cf-hash-full.pdf`, 2009.
5. Matteo Centenaro, Riccardo Focardi, Flaminia L. Luccio, and Graham Steel. Type-based analysis of pin processing apis. In *Proocedings of the 14th European Symposium on Research in Computer Security (ESORICS)*, volume 5789 of *Lecture Notes in Computer Science*, pages 53–68, Saint-Malo, France, September 21-23 2009. Springer.
6. Delphine Demange and David Sands. All secrets great and small. In *Programming Languages and Systems, 18th European Symposium on Programming (ESOP)*, volume 5502 of *Lecture Notes in Computer Science*, pages 207–221, York, UK, March 22-29 2009. Springer.
7. Riccardo Focardi and Matteo Centenaro. Information flow security of multi-threaded distributed programs. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, Tucson, AZ, USA, June 8 2008. ACM Press.
8. Peeter Laud. On the computational soundness of cryptographically masked flows. In George C. Necula and Philip Wadler, editors, *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 337–348, San Francisco, Ca, USA, January 10-12 2008. ACM Press.
9. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
10. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, January 2009.

11. Geoffrey Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125, Cape Breton, Nova Scotia, June 11-13 2001. IEEE.
12. Dennis Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (CSFW)*, pages 246–254, Cambridge, England, July 3-5 2000. IEEE.
13. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th Computer Security Foundations Workshop (CSFW)*, pages 156–169, Rockport, Massachusetts, USA, June 10-12 1997. IEEE Computer Society.

Appendix

A Case studies

In this section the case studies presented in the introduction are shown to type check and a new example will be introduced. Note that some syntactic sugars which were given introducing the examples has been removed here in order to show fully typed codes.

A simplified *su* command The first example is a simplified version of the *su* Unix utility. Let `root_shell` be a command which requires an high-integrity level to be computed, i.e., $\Delta \vdash \texttt{root_shell} : (\ell_C\mathsf{H}, t, f)$. The user entered password `t_pwd` will be deemed to be secret and low-integrity, i.e., $\Delta(\texttt{t_pwd}) = \mathsf{PH_bL}$. `root_passwd`, instead, stores the digest of the administrator password and it will be a high-integrity data since it is supposed to be stored in a write-protected file, let $\Delta(\texttt{root_passwd}) = \mathsf{D}\ell_C\mathsf{H}$. Note that the array notation have been replaced by a single variable, this does not affect the aim of the example which is proving the security of the Unix implementation of the password-based authentication mechanism.

```
trial := hash(t_pwd);
if (trial = root_passwd) then
    root_shell;
else
    FAIL;
```

The password is supposed to be strong thus it has been typed as a big secret. If such an assumption is removed, the code does not type, indeed the above program could be used to mount a brute-force attack on the password. The type system prevents such fact by requiring that the confidentiality level of the guard is at most a big secret in rule (int-test).

The confidentiality level of $\texttt{root_passwd}$ could be either setted to L or $\mathsf{H_b}$. This models the fact that having strong passwords, they could be safely stored in a public location.

Let $\Delta(\texttt{trial}) = \mathsf{DH_bL}$, the expression $\mathsf{hash}(\texttt{pwd})$ is typed $\mathsf{DH_bL}$ by rule (hash-b) and the first assignment is then typed $(\mathsf{H_bL}, \mathsf{LH}, \downarrow)$ by (assign). The if branch is typed $(\mathsf{H_bH}, \mathsf{H_bL} \sqcup t, \uparrow)$ by (int-test): if $\ell_C = \mathsf{L}$ by subtyping $\texttt{root_passwd}$ will be typed $\mathsf{DH_bH}$ while if $\ell_C = \mathsf{H_b}$ nothing special is needed. The sequential composition of the two commands is then typed by (seq-1), indeed $\mathsf{LH} \sqsubseteq \mathsf{H_bL}$.

Software Distribution A software company distributes an application using different mirrors on the Internet. Having downloaded the program from one of the mirrors, a user will install the given binary only if its digest matches the one of the original application provided by the software company.

```
if (hash(my_blob.bin) = swdigest) then
    trusted_blob.bin := my_blob.bin;
    install := 1;
else
    FAIL;
```

Let $\texttt{my_blob.bin}$ be the variable storing the downloaded binary, it is a low-integrity public variable, i.e., $\Delta(\texttt{my_blob.bin}) = \mathsf{PLL}$. The trusted digest given by the software company is stored in the $\texttt{swdigest}$ variable which is a high-integrity one ($\Delta(\texttt{swdigest}) = \mathsf{DLH}$). The installation of the application is simulated by first saving the low-integrity binary in the trusted location $\texttt{trusted_blob.bin}$ ($\Delta(\texttt{trusted_blob.bin}) = \mathsf{PLH}$) and then by assigning 1 to the high-integrity variable $\texttt{install}$ ($\Delta(\texttt{install}) = \mathsf{PLH}$).

The if branch types $(\mathsf{LH}, \mathsf{LL}, \uparrow)$ by (int-hash), indeed all the requirements on variables are satisfied by letting $\ell_C = \mathsf{L}$ in the typing rule and, the assignment to $\texttt{install}$ types $(\mathsf{LH}, \mathsf{LH}, \downarrow)$.

A new case study is now introduced, it shows a program which let a system administrator to manage the password file.

A simplified *passwd* This example presents a password update utility. It is a simplified version of the *passwd* Unix command where we require that only the administrator can perform such a task. This is due to the fact that the integrity of the password file must be preserved.

Three parameters are expected: the administrator password and the user old and new passwords.

```
root_trial := hash(t_root);
user_trial := hash(old);
if (root_trial = root_passwd) then
    if(user_trial = user_passwd) then
        user_passwd := hash(new);
    else
        FAIL;
else
    FAIL;
```

Variable `root_passwd` stores the digest of the root password while `user_passwd` the hash of the user one. These model, as in the first example, the needed portions of the password file ($\Delta(\texttt{root_passwd}) = \mathsf{DH_bH}$ and $\Delta(\texttt{user_passwd}) = \mathsf{DH_bH}$). The typed root password `t_root` is low-integrity, i.e., $\Delta(\texttt{t_root}) = \mathsf{PH_bL}$ as well as the `root_trial` variable used to store its digest ($\Delta(\texttt{root_trial}) = \mathsf{DH_bL}$). Similarly, $\Delta(\texttt{old}) = \mathsf{PH_bL}$ and $\Delta(\texttt{user_trial}) = \mathsf{DH_bL}$. The `new` variable which stores the new user password must be regarded as high-integrity ($\Delta(\texttt{new}) = \mathsf{PH_bH}$). This time the user input will be considered trusted since the intended user will be authenticated and only in that case the new password will be used. This is the only way to make the hash operator to type at a high-integrity level when storing the digest of the new password to `user_passwd`. In fact, there is no way to prove the integrity of a fresh new password by equality test.

The first two assignments type $(\mathsf{H_bL}, \mathsf{LH}, \downarrow)$ by (assign). The innermost if branch types $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$ by (int-test): variable `user_passwd` types $\mathsf{DH_bH}$ and `user_trial` $\mathsf{DH_bL}$, the assignment to `user_passwd` types $(\mathsf{H_bH}, \mathsf{LH}, \downarrow)$ by (assign) and the hash expression is typed by DLH by (hash-b) and promoted by subtyping to $\mathsf{DH_bH}$. In a similar way the main if branch is again typed by (int-test) obtaining $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$.

The whole program is thus typed $(\mathsf{H_bH}, \mathsf{H_bL}, \uparrow)$ by (seq-1).