IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 24 January 2024, accepted 13 February 2024, date of publication 21 February 2024, date of current version 4 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3368453

RESEARCH ARTICLE

How to Formalize Loop Iterations in Cryptographic Protocols Using ProVerif

TAKEHIKO MIENO[1], (Member, IEEE), HIROYUKI OKAZAKI[2], KENICHI ARAI[3], AND YUICHI FUTA[4]

[1]Business Development Division, EPSON AVASYS Corporation, Shinshu University, Nagano 386-1214, Japan
[2]Graduate School of Science and Technology, Shinshu University, Nagano 380-8553, Japan
[3]School of Information and Data Sciences, Nagasaki University, Nagasaki 852-8521, Japan
[4]School of Computer Science, Tokyo University of Technology, Tokyo 192-0982, Japan

Corresponding author: Takehiko Mieno (mieno.takehiko2@exc.epson.co.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 22K11982.

ABSTRACT The formal verification of cryptographic protocols has been extensively studied in recent years. To verify the cryptographic protocol security, formal verification tools consider protocol properties as interactive processes involving a cryptographic functionality. In general, we formally define a cryptographic functionality as an abstract function. However, the actual cryptographic protocols used comprise complex combinations of cryptographic functionalities. Thus, we may not determine if the protocol is secure, even when the cryptographic protocol security with cryptographic functionalities is verified using verification tools. Increasing the reliability of the verification results necessitates the verification of the security properties of these algorithms using ProVerif. Many cryptographic algorithms have been designed as iterative algorithms. These include the Merkle and Damgård construction (MD construction) method for cryptographic hash functions and the cipher block chaining mode (CBC mode) for the block cipher mode of operation. However, formalizing an iterative execution is difficult in ProVerif. Thus, we propose a method for formalizing a model of iterative executions. In the proposed method, we describe to formalize iterative execution as a formal model of function calls. We demonstrate the validity of our proposed method by formally verifying the safety of the MD construction method, which behaves like an iterative execution by including a one-way compression function and internal variables changed by the output of these functions. We also present a method for formalizing a common block cipher mode of operation (i.e., CBC mode) used in the proposed method to handle iterative execution.

INDEX TERMS CBC mode, formal method, iterative execution, MD construction, ProVerif.

I. INTRODUCTION

The formal verification of cryptographic protocols has been extensively studied in recent years. ProVerif [1], [2], [3] is one of the most successful automatic cryptographic protocol verifiers. ProVerif and similar verification tools, e.g., Tamarin Prover [4], [5], [6], Scyther [7], Verifpal [8], and AVISPA [9] can automatically analyze the cryptographic protocol security formally described in the Dolev and Yao model [10].

To verify the cryptographic protocol security, these tools consider the cryptographic protocol properties as interactive processes involving a cryptographic functionality. A formal model of a protocol comprises ideal cryptographic functionalities, such as secret key cryptographic functions, digital signatures, and cryptographic hash functions. In general, we formally define a cryptographic functionality as an abstract function. The actual cryptographic protocols in use; however, comprise complex cryptographic algorithm combinations; thus, we may not determine if the protocol is secure, even when the cryptographic protocol security with cryptographic

The associate editor coordinating the review of this manuscript and approving it for publication was Peter Langendoerfer.

functionalities may be verified using these tools. ProVerif verifies the security of formally described protocols constructed with an ideal cryptographic functionality, although actual protocols are implemented with feasible cryptographic algorithms. To increase the reliability of the verification results, it is desirable to verify the security properties of such algorithms using ProVerif. Many cryptographic algorithms are designed as iterative algorithms, including the Merkle–Damgård construction (MD construction) [11], [12] method for cryptographic hash functions and the cipher block chaining mode (CBC mode) [13], [14], [15], [16] for the block cipher mode of operation. We propose herein a method for formalizing a model of iterative executions. However, formalizing the iterative execution is difficult because in the usual method, we can only formalize the cryptographic communication procedure between processes when ProVerif is used to verify the cryptographic protocol security.

Our proposed method formalizes function calls by treating them as a communication between internal processes. We formalize the MD construction algorithm using this method and verify that the formal MD construction model holds the cryptographic hash function properties. MD construction is a method of constructing a cryptographic hash function (e.g., SHA–1 [56] or MD5 [57]). The MD construction behavior is similar to that of the iterative executions because the MD construction includes one-way compression functions and an internal variable changed by the output of these functions. We also present a method for formalizing a common block cipher mode of operation, i.e., the CBC mode. A block cipher is regarded as a secure cryptographic transformation used to encode/decode one fixed-length bitstring referred to as a block. The operation mode typically behaves as an iterative execution.

This paper discusses how to deal with an iterative execution in ProVerif. The main approach involves the internal states being described by the communication between the processes via private channels. Please see Section III-A for more information on the private channels. We propose a method for formalizing the loop iterations for ProVerif, a software that verifies the cryptographic protocol security, and confirms the correctness of the proposed formalization through case studies of the Merkle–Damgård structures of hash functions and the CBC mode of block ciphers.

The remainder of this paper is organized as follows: Section II provides an overview of the formalization performed using the model validation tool, ProVerif; Section III describes our proposed iterative formalization method using S/KEY; Sections IV and V present the formal verification of the MD construction and the CBC mode using the proposed method for iterative executions; and Section VI discusses our proposal and its limitations and the derivation of the padding oracle attack [44], [45] against the CBC mode. Note that Sections III, IV, and V are all formalized using our proposed method to handle the iterative execution.

A. MOTIVATIONS AND CONTRIBUTIONS

The main contribution of this work is our proposed method that allows the iterative method formalization. The proposed method formalizes the function calls by treating them as communication between internal processes using a private channel. It allows the iterative method formalization used in complex cryptographic protocols other than MD construction and the CBC mode.

Notably, however, because ProVerif is a model checking tool, its verification result is not a proof of security, even though it may find a feasible attack via an exhaustive search of the given model and rules.

1) HOW TO FORMALIZE AN ITERATIVE EXECUTION

Our method formalizes function calls by treating them as communications between internal processes through a private channel. Our approach enables the formalization of the iterative methods used in the complex cryptographic protocols in ProVerif, showing application to the design of other cryptographic algorithms. The proposed method uses the descriptions of internal communication with self-duplicating processes in ProVerif. The internal states with communication between processes are described through private channels. This concept is similar to the interprocess communication in general computational models. Our method enables the formalization and the security verification of cryptographic modules.

2) CASE STUDIES

We confirm herein the validity of the proposed formalization through case studies on the Merkle–Damgård structures of hash functions and the CBC mode of block ciphers. We demonstrate the formal verification of the MD construction and the CBC mode using the proposed method for iterative executions[1]. We verify that the proposed method is consistent with the formal MD construction model and satisfies the cryptographic hash function properties of pre-image resistance (PR), second-preimage resistance (2ndPR), and collision resistance (CR). We also formalize the CBC mode with and without padding and verify that we can encrypt/decrypt a plaintext comprising multiple blocks in the CBC mode while keeping the CBC mode communication a secret.

The proposed formalization can discover the vulnerabilities [19], [20], [21], [22] imposed by the structure of the hash functions and encryption modes, such as length extension attack (LEA) [23] and chosen plain attack (CPA) [55]. In a formal verification, the hash functions and block ciphers are formalized as ideal functions without vulnerabilities. We illustrate herein the validity of our formalization and validation results by formalizing the LEA on iterated hash functions (Section IV-B4), MD-strengthening [42], [43]

[1]Note that the MD construction formalization has already been published in international conferences [17] and [18]. This paper is the final paper that summarizes these results.

method that makes the attack impossible (Section IV-B6), and CPA on the CBC mode (Section V-B5).

B. RELATED WORK

Blanchet and Paiola [24] presented a novel automatic technique of proving the secrecy and authentication properties of security protocols manipulating the lists of unbounded length over an unbounded number of sessions. This approach requires the extension of the Horn clause approach. Another method using a tamarin theorem prover was introduced [5], [25]. To the best of our knowledge, ProVerif in iterative processes has not been previously proposed. Conversely, Backes et al. [26] formally proved the CR of MD construction using EasyCrypt [27], a theorem prover tool that supports the validation of mathematical theorems and specializes in proving the cryptographic protocol security. By contrast, ProVerif is a model checker, not a theorem prover. ProVerif automatically and exhaustively searches the execution processes of a protocol and is used to discover detailed cryptographic protocol attacks. Thus, an MD construction formally using the EasyCrypt result cannot be applied to ProVerif.

In recent years, ProVerif has become a popular tool [28] for protocol security verification. ProVerif particularly identifies attacks on various protocols, including Extend ProVerif [29], Bluetooth protocol design vulnerabilities [30], distance bounding protocols [31], TLS1.3 [32], on-site voting systems [33], authentication protocols for Internet of Drones environments [34], Fast Identity Online 2 [35], and the MobileRFID authentication protocol [36]. Jacomme et al. [37] found weaknesses in their EDHOC analysis, a key exchange proposed by the IETF's Lightweight Authenticated Key Exchange Working Group. Cheval and Rakotonirina [38] applied two complex industrial-scale voting protocols designed for the Swiss context. The SAPIC+ [39] protocol verification platform allows the transparent use of three major verification tools. Boyd and Anish [40] summarized the cryptographic protocols and their securities. Barbosa et al. [41] summarized the formal verification of cryptographic protocols.

II. PROVERIF

ProVerif is an automatic cryptographic protocol verifier in the formal Dolev–Yao model. It assumes that cryptographic functionalities are ideal and is based on an abstract representation of the protocol by Horn clauses [46]. ProVerif can be used to verify the cryptographic properties of secrecy [47], authentication [48], strong secrecy [49], weak secrecy [50], and observational equivalence [50]. The ProVerif input file comprises the declaration and execution parts.

The terms and components of cryptographic protocols (e.g., variables, functions, and channels) are defined in the declaration. We also define processes, rewrite term rules, and include queries in the declaration. Note that attackers can use nonprivate terms in the verification phase. We define terms to construct processes. To define the protocol to be verified, we describe how the process is executed in the execution part.

A. PROCESS EXECUTION SYNTAX

This section presents a brief review of the execution. We demonstrate the execution state of the protocol in the execution part. We directly describe the initiator and responder processes defined in the declaration part, which are executed in parallel as follows:

$$Initiator \mid responder_2 \mid responder_1 \mid \cdots .$$

!P represents the unbounded number of replications of P, which are executed in parallel as follows:

$$P \mid P \mid P \mid \ldots .$$

Processes communicate through a public channel. An attacker can intercept messages through these channels and send any message to the channel. Note that we omitted the declaration details. Please refer to [2] for the declaration details.

B. PROVERIF QUERIES

This section introduces essential queries for the ProVerif verifier. ProVerif has some queries that are directives to the verifier to determine whether the protocols possess the designated security properties. We can check basic security requirements, such as secrecy and authentication, using these designated queries. However, the designated queries of ProVerif are insufficient for the security verification of advanced security requirements. The proposed method allows us to check more detailed advanced security requirements by combining the queries and the processes indicating the attack goal. In this paper, the basic designated queries are summarized as follows:

Secrecy: "query attacker (X)"
: The verifier checks whether the attacker can know term X. If "not attacker (X)" is true, term X is kept secret until the protocol terminates.

Reachability: "query event (X)"
: The verifier checks the reachability of event X. Event X is not executed if "not event (X)" is true. Event X can be executed if the query is false.

C. GENERAL APPROACH TO FORMALIZING PROTOCOLS IN PROVERIF

In this section, we briefly review the general approaches used to formalize the protocols in ProVerif. We explain herein the formalization of a protocol as communication between processes (e.g., sender and receiver processes) in ProVerif. Code.1[2] presents a simple protocol (Figure 1).

```
  (* Code.1: EXAMPLE of Protocol *)
1   free c:channel.
2   free m:bitstring[private].
3
4 Let Sender =
```

[2]In this study, all codes were verified using ProVerif 2.05.

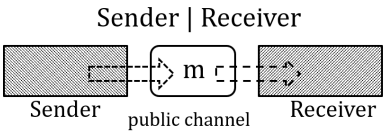

FIGURE 1. Example of the general protocol.

```
5     new n:bitstring;
6     out (c,m).
7
8   let Receiver =
9     in (c, r:bitstring).
10
11  process
12    Sender | Receiver
```

In the declaration part of Code.1, we declared terms `c` and `m` and defined the `Sender` and `Receiver` processes. Here, "`free`" denotes the declarator of a global free term, whereas "`new`" denotes the declarator of a local term in individual processes.

Channel type terms are treated as a set in ProVerif and initially declared as an empty set. Term `c` is declared as a channel type term. Term `m` is declared as a bitstring type term with a private attribute. Term `n` is declared as a local term in the `Sender` process when the `Sender` is executed. Note that the `Sender` and `Receiver` processes are executed in parallel in the execution part. The executed processes communicate with each other by sending and receiving terms via the indicated channels. In this case, `Sender` sends `m` to channel `c`, and `Receiver` receives the term from channel `c`.

In ProVerif, attackers can know the global terms without private attributes, but initially cannot know these with private attributes and local terms. However, any term sent to channels without a private attribute is exposed to attackers, even if it is initially private. The subsequent section discusses the technical details of the ProVerif channels.

Here, term `m` is inserted in the set `c` when `out(c,m)` is executed in the `Sender` process. The local term `r` is declared and initialized with one of the terms in `c` when `in(c, r:bitstring)` is executed in the `Receiver` process. Note that it cannot remove terms from `c`, even if the processes receive terms from `c`.

However, we can declare multiple channel type terms to separate the communication channels. Here, the terms sent to a channel, which are declared with private attributes, are kept secret from the attackers. The next section describes a method for formalizing a function call with a private channel in ProVerif.

III. FORMALIZING AN ITERATIVE EXECUTION IN PROVERIF

Although actual protocols are implemented with feasible cryptographic algorithms, ProVerif generally verifies the security of formally described protocols constructed with an ideal cryptographic functionality. To increase the reliability of verification results, it is desirable to verify the security properties of such algorithms using ProVerif. However, several cryptographic algorithms, including iteration and recursion ones, are not supported by ProVerif.

In general, an iteration can be formalized by iteratively and explicitly enumerating the same computations. However, this approach may be inconvenient for the security verification of practical protocols. We propose herein a method for formalizing an iterative execution as a formal function call model.

We introduce the proposed method for formalizing function calls, specifically focusing on the iterative execution formalization in ProVerif. We consider the S/KEY (III-A), MD construction (IV-B2), and CBC mode (V-B1,V-B2,V-B3) as examples.

Our proposed method allows the formalization of iterative and recursive executions. Note that ProVerif may not terminate during verification because it works for an unbounded number of sessions and an unbounded message space. (Please refer to [2].) This problem is mainly caused by the model conditions and can be avoided by properly considering them. For clarity, we explain the fixed-bound case (i.e., set the value to fixed conditions) in this work.

A. BASIC APPROACH FOR FORMALIZING ITERATIVE EXECUTIONS

This section describes the proposed method for formalizing function calls by treating them as a communication between internal processes through a private ProVerif channel.

Function calls and subroutines can be realized by the internal communication processes between the main process and other processes. Iterative executions can repeatedly be formalized as an internal interprocess communication. We claim that these calculations can be formalized as an internal interprocess communication, even if the actual calculations do not necessarily match the expressed model.

This concept requires a consideration of the formal channel model characteristics presented in Section II-C.

We introduce herein a formalization of the calculation of a one-time password as an example of a simple iterative execution (Figure 2). Code.2 depicts an example of the formalization of an iterative execution calculating a hash function for a specified number of times. In this example, the main process invokes the `SKEY` process to calculate the hashing of the given seed repeatedly three times. Consequently, the `SKEY` process computes the `hash(hash(hash(seed)))` via an iterative execution.

```
  (* Code.2: One-time Password *)
1   free t1:channel[private].
2   free t2:channel[private].
3   free seed:bitstring [private].
4
5   fun hash(bitstring):bitstring.
6
7   event COL.
8   query event(COL).
9
10  let SKEY(s:bitstring) =
11    in(t1, x2:nat);
```

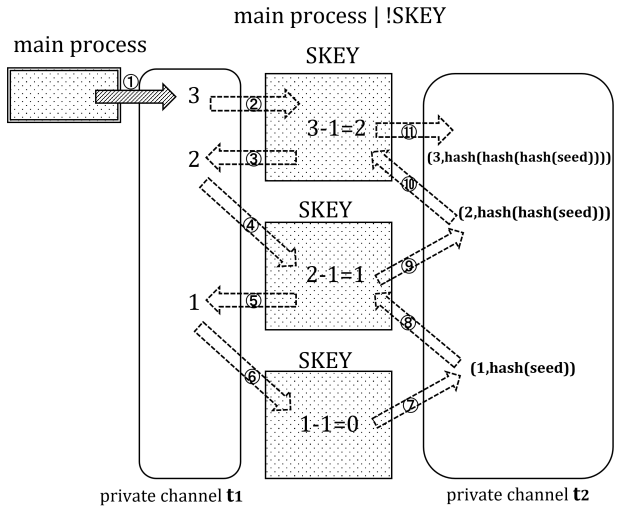

FIGURE 2. Formalization of an iterative execution.

```
12    let x3:nat = x2-1 in
13    if (x3 <> 0) then
14    (
15      out(t1, x3);
16      in(t2,(x4:nat,sk:bitstring));
17      if(x4=x3) then
18      (
19        let otp3 = hash(sk) in
20        out(t2,(x2,otp3));
21        if(otp3=hash(hash(hash(seed)))) then
22        event COL
23      )
24    )
25    else
26      out(t2, (x2,hash(seed))).
27
28  process
29    let a1:nat = 3~in
30    !(
31      out(t1, a1)
32    )
33    | !SKEY(seed)
```

In lines 10–26 of Code.2, the `SKEY` process is a formal model representing one step of the hash iterative execution. In line 33 of Code.2 in the execution part, "`!SKEY(seed)`" implies the execution of an unbounded number of replications of the `SKEY` process. During this execution, "`!SKEY(seed)`" behaves as an iterative subroutine by communicating the `SKEY` replications with each other via an internal channel. In this case, "`seed`" denotes an argument representing the shared secret key that each replication of the `SKEY` process takes at runtime. First, the main process invokes `SKEY` as a subroutine by sending the nat term `a1` representing the number of replications via a private channel `t1` in line 31 of Code.2: "`out(t1,a1)`" (`a1 = 3` in this case).

A `SKEY` replication process communicates with the next step via private channels `t1` and `t2`, where terms `t1` and `t2` denote the channels for the counter of repetitions and for the temporary data during iterative execution, respectively. The replication process of `SKEY` receives the nat term `x2` via `t1` in line 11 of Code.2. Subsequently, `SKEY` computes `x3 = x2-1`. In the case of `x3 = 0`, `SKEY` sends `(x2,hash(seed))` to return to the previous step of `SKEY` via `t2`.

In the case of `x3 ≠ 0`, `SKEY` sends `x3` to another replication of `SKEY` via `t1` for an iterative execution. The current `SKEY` then receives a pair of terms `(x4,sk)` via a private channel `t2` from another `SKEY` replication. The current `SKEY` must ensure that the received data come from the next step of `SKEY` considering the formal model characteristics of channels (Section II-C). For `x4 = x3`, `(x4,sk)` denotes return from the next step. Thus, the current `SKEY` returns `(x2,hash(sk))` to the previous `SKEY` step or the main process via channel `t2`.

Consequently, the "`!SKEY`" process can now compute a given seed hashed for a specified number of times via iterative executions. Lines 21 and 22 of Code.2 are for the query to confirm if this sample works well. They are irrelevant to the formalization of the iteration behavior. In ProVerif, "`out(t1,a1)`" means outputting term `a1` to channel `t1`. In this case, `a1` indicates the number of times the hash function is applied. It will be sufficient to the term `a1` once if `t1` is a public channel. In our formalization, however, `a1` must be repeatedly output if `t1` is a private channel. This difference is caused by how ProVerif handles models for public and private channels [2].

The terms output to public channels are expected to be referenced by anyone, including attackers; thus, the term remains in the public channel, even if it is input to a process. By contrast, the terms output to private channels vanish from these channels if they are input to a process. Therefore, when multiple processes refer to a term in a private channel, the same valued term must repeatedly be output to this private channel.

As described above, the iterative process of ProVerif exhibits different behaviors in public and private channels. The subsequent section provides a detailed explanation for this (III-B).

B. DETAILS OF PUBLIC AND PRIVATE CHANNELS WITH ITERATIVE EXECUTIONS

This section describes the iterative process of ProVerif that exhibits different behaviors in public and private channels.

`out(t1,a1)` outs `a1` (= 3) are output to the private channel `t1`. Parameter `a1` indicates the number of hash applications. A public channel requires one `out(t1,a1)` execution, whereas a private channel needs multiple `out(t1,a1)` (`!out(t1,a1)`) replications. The reason for this is explained below. In the ProVerif channel, when one wants to put a term from the channel, the term needs to be out to the channel in advance. (If a channel is regarded as a set, when a certain term is out, this term is added to the set. When it is in, the term is taken out from the set). For a public channel, the term out to the channel is available to the attacker; hence, the attacker can use it to exit the channel at any time. In the case of public communication channels,

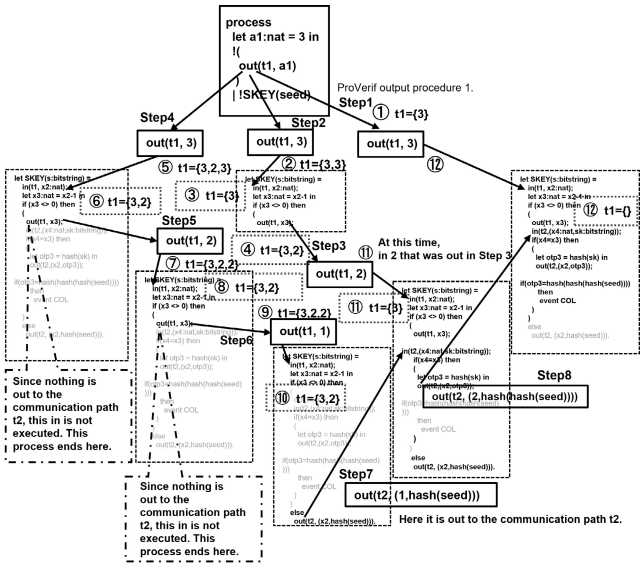

FIGURE 3. Output procedure "! out(t1,a1)".

"! (out(t1,a1))" : Changes in values in private channels t1 and t2.

No	direction	Channel : t1	Channel : t2	supplement
① Step1		{3}		t1:{3} -> in ⑫
② Step2	out	{3, 3}		
③	in	{3}		
④ Step3	out	{3, 2}		t1:{2} -> in ⑪
⑤ Step4	out	{3, 2, 3}		
⑥	in	{3, 2}		
⑦ Step5	out	{3, 2, 1}		
⑧	in	{3, 2}		
⑨ Step6	out	{3, 2, 1}		
⑩	in	{3, 2}		
⑪ step7	out		{1, hash(seed)}	
	in	{3}		
⑫ step8	out		{1, 2,hash(hash(seed))}	
	in	{}		

FIGURE 4. Transition table of private channels t1 and t2.

the attacker can execute `out(t1,a1)` many times by executing `out(t1,a1)` once; therefore, the replication of `out(t1,a1)` is not necessary. However, the attacker cannot intervene in a private channel; thus, our proposed `SKEY` model (Figure 2) requires the replication of `out(t1,a1)`.

"Figure 3" shows the "`in`" and "`out`" flows in the case of "`!out(t1,a1)`." In the proposed `SKEY` model, `out(t1,a1)` is executed only once. If we wish to insert the natural number 2 in "Figure 3"(⓫) and the natural number 3 in "Figure 3"(⓬), then "`in`" cannot be performed. In contrast, `out(t1,a1)` can be executed if it is to be replicated, as shown in ⓫ and ⓬ in "Figure 3." Therefore, our `SKEY` model (Section III-A) requires a replication of `out(t1,a1)`. "Figure 4" is a transition table of values in a private channel (`t1,t2`).

IV. FORMALIZATION OF THE MD CONSTRUCTION

A. MD CONSTRUCTION

A cryptographic hash function H transforms a message of any length to a bitstring of fixed-length. The hash function must possess security properties, such as PR, 2ndPR, and CR.

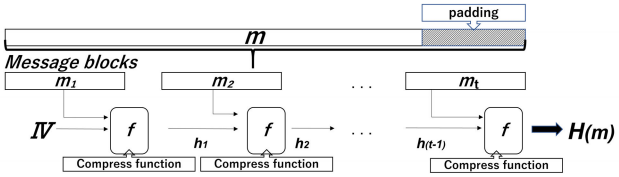

FIGURE 5. MD construction.

The MD construction [11], [12] is a method for constructing a cryptographic hash function, for example, SHA-1 or MD5 (Figure 5).

"Figure 5" shows the MD construction mechanism, where an arbitrary-length input message (m) is divided into fixed-length blocks (m_1,m_2,...,m_t) sequentially input to a fixed-length compression function f in order from the beginning block of the input message. The final output (h_t) used as a hash value is obtained as follows:

$$m = (m_1, m_2, \ldots, m_t) \tag{1}$$
$$h_0 = IV \tag{2}$$
$$h_i = f(h_{i-1}, m_i) \quad for \quad i = 1, 2, \ldots, t \tag{3}$$
$$h_t = H(m). \tag{4}$$

IV denotes the initial vector. If the final block is shorter than the block length, padding is added such that it becomes the block length. Unfortunately, ProVerif does not support the function call formalization. In our formalization, the state of the process using a table, which is a data structure supported by ProVerif, is maintained to reproduce the intended behavior of the process' control structure. The proposed formalization method controls internal process communications without relying on a table structure.

B. FORMALIZATION OF THE MD CONSTRUCTION CODE

Code.3[3] shows the formalization of the MD construction.

```
(* Code.3: Formalization of MD Construction *)
1   free c:channel.
2   free s:channel[private].
3   free t:channel[private].
4   free m:bitstring[private].
5   const iv:bitstring.
6
7   fun con(bitstring,bitstring):bitstring.
8   fun divhead(bitstring):bitstring.
9   fun divrest(bitstring):bitstring.
10  equation forall mt:bitstring;
11    con(divhead(mt),divrest(mt))=mt.
12
13  (* Compress function *)
14  fun comp(bitstring,bitstring):bitstring.
15
16  (* Query *)
17  event PROOF.
18  query event(PROOF).
19
20  (* MD construction *)
21  let makeMD =
22    in(s,(ha:bitstring,rm:bitstring,bl:nat));
23    if(bl <> 1) then
24    (
25      let newbl = divhead(rm) in
26      let newstream = divrest(rm) in
```

[3]Improved version from [17] and [18].

```
27      let newha = comp(ha,newbl) in
28      out(s,(newha,newstream,bl-1))
29    ) else (
30      let MDh = comp(ha,rm) in
31      out(c,MDh);
32      event PROOF
33    ).
34
35  (* Main process *)
36  process
37    (
38      out(s,(iv,m,4)) | !makeMD
39    )
```

1) MESSAGE BLOCK DIVISION

This section describes how the message block division is formalized. Note that our formalization only supports message division into a predetermined number of blocks, as mentioned in Section IV-A. In Code.3, we treat a message as a bitstring type term. Unfortunately, a free term just declared is a symbol without information other than its name and type, although we divide an arbitrary-length message into fixed-length blocks in the real world. We formalize splitting a message into two blocks as follows. First, two bitstring terms are generated from the given bitstring term. We then introduce the following relationship among the three bitstring terms: the given term is equal to the concatenation of the two generated terms.

Accordingly, we need to provide declared terms with information about length and the division procedure. We handle the length information of a bitstring term by introducing an additional nat type term representing the bitstring length. We then introduce the relation of how to divide a bitstring term through bitstring division and concatenation functions.

The "`con`" function defined in line 7 of Code.3 generates the concatenation of two bitstring type terms. The `divhead` and `divrest` functions in lines 8 and 9 of Code.3 extract a portion from the bitstring. Note that these functions are treated as an argument. Here, `divhead` extracts the first block of the given bitstring, whereas `divrest` extracts a portion other than what `divhead` extracts. For example, let X be a bitstring type term and ℓ be a nat type term representing the X length. In this case, the bitstring term X is divided into two bitstrings, that is, `divhead(X)` and `divrest(X)`, where `con(divhead(X),divrest(X)) = X`. In this model, we treat the bitstring lengths `X`, `divhead(X)`, and `divrest(X)` as "ℓ, 1, $\ell - 1$." We need to keep the block length of two bitstrings as individual nat terms after division because `divhead` and `divrest` generate a bitstring type free term. We define the rewriting rules for concatenation in lines 10 and 11 of Code.3 and formalize the binary splitting of the bitstring message in this manner. Repeating this process enables us to formalize the dividing message into any number of blocks.

2) ITERATIVE EXECUTION OF THE MD CONSTRUCTION

As discussed in Section IV-A, the MD construction algorithm iterates compression function computations that take the output of the previous round and the divided message as the input. We propose herein a formal model that represents the iterative execution of the MD construction rounds. The `makeMD` process in lines 21–33 of Code.3 depicts a formal model representing each round of the iterative execution of the MD construction. On the right side of line 38 of Code.3 in the main process, "`!makeMD`" represents invoking an unbounded number of `makeMD` process replications.

We formalize the iterative execution of the MD construction rounds as the internal process communication of the `makeMD` replications. First, the main process invokes `makeMD` as a subroutine by sending triad bitstrings via the private channel `s` in line 38 of Code.3 as follows: `out(s,(iv,m,4))`, where `(iv,m,4)` is a message with a length divided by "`4`", and "`iv`" is a constant bitstring term representing the initial vector (*IV*). Once the main process invokes the first step of `makeMD`, each `makeMD` process replication starts to communicate with the next replication via the private channel.

Consequently, the iterative executions of `makeMD` may now compute the MD construction. We formalized the division of a message into a predetermined number of blocks during the MD construction in ProVerif. To ensure that this model functions as intended, we verified the reachability of the `event PROOF` defined in lines 17 and 18 of Code.3, where MD construction was successfully completed by ProVerif. The ProVerif verifier found the execution path from the initiation of the execution to the `event PROOF`. The proposed method allowed us to formalize the subroutine call, especially the iterative executions. Note the possibility of declaring a channel with the "private" attribute in ProVerif. Any term sent via a private channel is secretly kept from attackers. Our proposed model handles a private channel as an internal communication between local processes.

The replication process of `makeMD` receives bitstring terms `(ha,rm,bl)` via the secret channel in line 22 of Code.3. Here, `ha` is the compress function output in the previous step of `makeMD`, and `rm` is the preimage tail bitstring. "Figure 6" shows an example of the block division process in the MD construction.

If `bl ≠ 1`, `makeMD` divides `rm` into two bitstring terms, that is, `newbl = divhead(rm)` and `newstream = divrest(rm)`.

Next, the compress function comp takes `newbl` and `ha` as arguments, generates "`newha = comp(ha,newbl)`," and then sends `(newha,newstream,bl-1)` to the next step of `makeMD` via the private channel. If `bl = 1`, the current step is the final step of the MD construction execution. Subsequently, `makeMD` returns `comp(ha,newbl)` as the MD construction calculation result. The `makeMD` process represents a single round of MD construction in the formalized MD construction model. That is, it does not formalize the entire MD construction process. We duplicated the `makeMD` process ("`!makeMD`") by iteratively executing the `makeMD` process, thereby effectively formalizing the entire MD construction process.

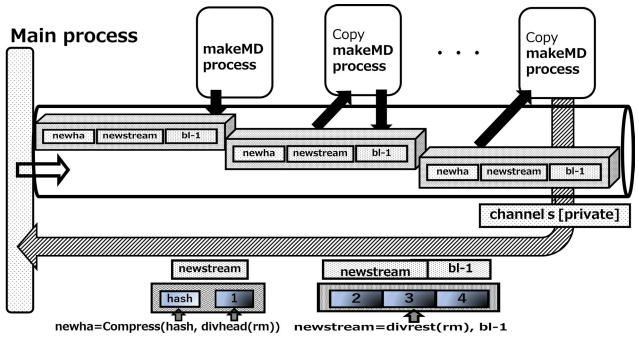

FIGURE 6. Block division process.

3) COLLISION RESISTANCE

Collision resistance makes it hard to find two different messages (i.e., M and M'), such that "$H(M) = H(M')$." Unfortunately, ProVerif does not have queries to directly verify the CR. The process described below is performed to present the verification goal to ProVerif and confirm the intentional collision resistance by ProVerif. The CR property complicates the search for different message sets with equal hash values. The CR can be verified by rewriting line 16 and onward in Code.3. In line 20 of Code.4[4] `table MD` means, need to be associated, which the original message and the generated hash value, it's uses to values from the table. To make it possible for the attacker to perform the MD construction, the `makeMDadv` process is described in lines 23–36 of Code.4 as the MD construction process for the attacker.[5]

```
  (* Code.4: Verifying Collision Resistance *)
16  (* Query *)
17  event COL.
18  query event(COL).
19
20  table MD(bitstring,bitstring).
21
22  (* MD construction for adversary *)
23  let makeMDadv =
24    in(t,(ha:bitstring,rm:bitstring,bl:nat,
25     mm:bitstring));
26    if(bl <> 1) then
27    (
28      let newbl = divhead(rm) in
29      let newstream = divrest(rm) in
30      let newha = comp(ha,newbl) in
31      out(t,(newha,newstream,bl-1,mm))
32    ) else (
33      let MDh = comp(ha,rm) in
34      insert MD(mm,MDh);
35      out(c,MDh)
36    ).
37
38  (* Indicating-Process for Collision Resistance *)
39  let Collision =
40    in(c,(m1:bitstring,m2:bitstring));
41    get MD(=m1,MDm1) in
42    get MD(=m2,MDm2) in
43    if(m1 <> m2 && MDm1 = MDm2)
44    then event COL.
45
46  (* Main process *)
47  process
```

[4]Improved version from [18].

[5]In general, the actual adversary is opposed to idealized ones referred to as attackers. The adversary is used in the coding of this work.

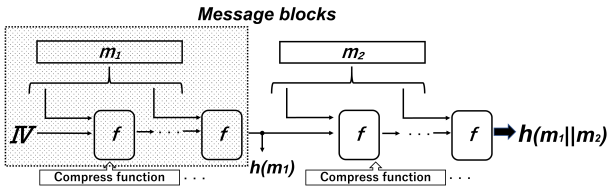

FIGURE 7. LEA.

```
48    !(
49    in(c,mm:bitstring);
50    out(t,(iv,mm,4,mm))
51    | !makeMDadv
52    )
53    | !Collision
```

The collision process in lines 39–44 of Code.4 verified whether an attacker generates two different inputs of the MD construction algorithm with equal output values (i.e., collision occurrence). As a result of the ProVerif verification, "true" was output, indicating that no successful attack path breaking the CR was detected. We formalized PR and 2ndPR in the same manner used in the proposed method. We confirmed that the verification method for the cryptographic functionalities is effective.

4) LENGTH EXTENSION ATTACK

Cryptographic hash algorithms comprising only elementary MD construction are vulnerable to the length extension attack (Figure 7). Thus, practical cryptographic hash functions based on the MD construction have been designed to withstand the LEA and other attack types.

We present herein how the ProVerif verifier discovered an execution path of the LEA against the pure MD construction algorithm to demonstrate the performance of our MD construction formalization. The LEA can be verified by rewriting line 16 and onward of Code.3.

```
  (* Code.5: Length Extension Attack *)
15  table MD_h(bitstring,bitstring).
16  table MD_h_check(bitstring).
17
18  (* Query *)
19  event SUCCESS.
20  query event(SUCCESS).
21
22  (* MD construction *)
23  let makeMD =
24    in(s,(ha:bitstring,rm:bitstring,bl:nat,
25     mm:bitstring));
26    if(bl <> 1) then
27    (
28      let newbl = divhead(rm) in
29      let newstream = divrest(rm) in
30      let newha = comp(ha,newbl) in
31      out(s,(newha,newstream,bl-1, mm))
32    ) else (
33      let MDh = comp(ha,rm) in
34      insert MD_h(mm, MDh)
35    ).
36
37  let LEA_CHECK =
38    in(c, MDh12':bitstring);
39    get MD_h_check(=MDh12') in
40    event SUCCESS.
41
42  (* Main process *)
43  process
```

```
44    (
45      new m1:bitstring;
46      new m2:bitstring;
47      out(c, (m2,2));
48      out(s,(iv,m1,2, m1));
49      get MD_h(=m1, MDh1:bitstring) in
50      out(c, MDh1);
51      out(s,(MDh1,m2,2, con(m1,m2)));
52      get MD_h(=con(m1,m2), MDh12:bitstring) in
53      insert MD_h_check(MDh12)
54    ) | !makeMD | !LEA_CHECK
```

In Code.5, we added the LEA detection mechanism of the LEA execution to Code.3. First, the main process of Code.5 computed h($m1$) and h($m1$|| $m2$) by invoking the `makeMD` process. h($m1$) and h($m1$|| $m2$) were registered in the private `MD_h_check` table, which cannot be accessed by attackers. The main and legitimate processes (e.g., `makaMD` and `LEA_CHECK`) were permitted to read and write to the `MD_h_check` table. The main process provided the attacker with the hash value registered in the table and determines whether the LEA succeeds.

The LEA succeeded when the `event SUCCESS` in the process `LEA_CHECK` was executed. Note that all functions comprising the `makeMD` process were permitted to use to any entity; thus, the attackers can perform a computation similar to when the process `makeMD` was used, even though the attacker was not allowed to directly use the process `makeMD`. The ProVerif verifier finds the execution path to the `event SUCCESS` if and only if both h($m1$) and $m2$ are given to the attacker; thus, in Code.5, the model formalized the MD construction, such that the LEA succeeded.

The "`LEA_CHECK`" process was defined in lines 37–40. ProVerif determined whether the LEA by the attacker is possible in line 54 "`!LEA_CHECK`." The query for "`event SUCCESS`" was defined in lines 19 and 20. In this formalization, "`event SUCCESS`" may be reachable. In other words, the attacker can execute the LEA. Here, the verification result by ProVerif output "false." ProVerif derived a specific attack procedure when the query "`event SUCCESS`" was executed. Appendix A presents the verification result of the LEA.

5) PREVENTION OF THE LENGTH EXTENSION ATTACK

Section IV-B4 demonstrated that our formal model of cryptographic hash functions based on elementary MD construction was as vulnerable to the LEA attacks as it is in the practical case. Understandably, practical cryptographic hash functions based on the MD construction were designed to withstand the LEA. In this section, we present a formalization of the cryptographic hashing algorithm that can avoid the LEA (Code.6). The formalization presented in Code.6 prevented the LEA attacks using the message length information in the final round of the MD construction. Note that practical cryptographic hash functions prevent the LEA using the MD-strengthening [42], [43] method that handles padding and message length information. The subsequent section presents the MD strengthening formalization.

```
(* Code.6: Length Extension Attack Prevention *)
15  fun length(bitstring):bitstring.
16
17  table MD_h(bitstring,bitstring).
18  table MD_h_len(bitstring,bitstring).
19  table MD_h_check(bitstring).
20
21  (* Query *)
22  event SUCCESS.
23  query event(SUCCESS).
24
25  (* MD construction *)
26  let makeMD =
27    in(s,(ha:bitstring,rm:bitstring,bl:nat,
28     mm:bitstring));
29    if(bl <> 1) then
30    (
31      let newbl = divhead(rm) in
32      let newstream = divrest(rm) in
33      let newha = comp(ha,newbl) in
34      out(s,(newha,newstream,bl-1, mm))
35    ) else (
36      let MDh = comp(ha,rm) in
37      insert MD_h(mm, MDh)
38    ).
39
40  let makeMDlen =
41    in(t,(ha:bitstring,rm:bitstring,bl:nat,
42     len:bitstring,
43     mm:bitstring));
44    if(bl <> 1) then
45    (
46      let newbl = divhead(rm) in
47      let newstream = divrest(rm) in
48      let newha = comp(ha,newbl) in
49      out(t,(newha,newstream,bl-1,len, mm))
50    ) else (
51      let newha = comp(ha,rm) in
52      let MDh = comp(newha,len) in
53      insert MD_h_len(mm, MDh)
54    ).
55
56  let LEA_CHECK =
57    in(c, MDh12':bitstring);
58    get MD_h_check(=MDh12') in
59    event SUCCESS.
60
61  (* Main process *)
62  process
63    (
64      new m1:bitstring;
65      new m2:bitstring;
66      out(c, (m2,2));
67      out(t,(iv,m1,2,length(m1), m1));
68      get MD_h_len(=m1, MDh1':bitstring) in
69      out(c, MDh1');
70      out(s,(iv,m1,2, m1));
71      get MD_h(=m1, MDh1:bitstring) in
72      out(t,(MDh1,m2,2,length(con(m1,m2)),
73      con(m1,m2) ));
74      get MD_h_len(=con(m1,m2), MDh12:bitstring) in
75      insert MD_h_check(MDh12)
76    ) | !makeMD | !makeMDlen | !LEA_CHECK
```

The length information to be added in line 15 of Code.6 was the message length, "`fun length(bitstring)`": A bitstring function was prepared, and the length was expressed by `length(m)`. Lines 40–54 defined the `makeMDlen` process, which added the length to the end of the message. $m2$ in line 66 and `comp`($m1$,`length`($m1$)) in line 69 were given to the attacker in public channel c. In the "`LEA_CHECK`" process (lines 56–59), under the abovementioned conditions, we verified whether the attacker can compute $h(m1||\ m2)$ = `comp`(`comp`($m1$,$m2$),

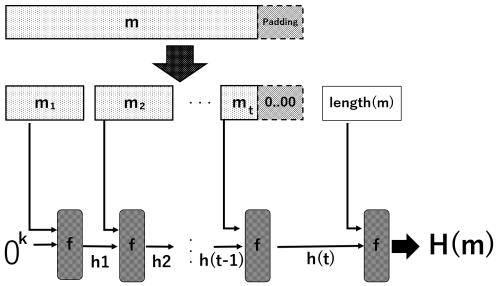

FIGURE 8. MD-strengthening.

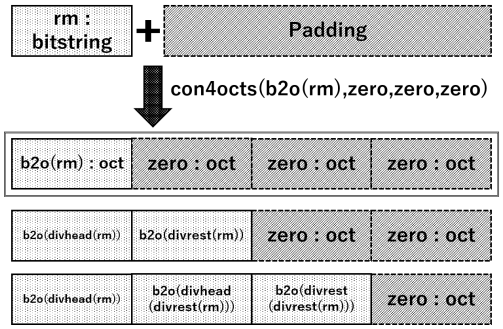

FIGURE 9. MD-strengthening padding process.

length($m1 \| m2$)). The verification result by ProVerif output "true," indicating that no successful LEA paths were found.

6) PREVENTION OF LENGTH EXTENSION ATTACK BY MD STRENGTHENING

MD strengthening is a padding method (Figure 8), in which a padding block with length information is added to the message. The security verification against LEAs for MD strengthening was essentially the same as the formalization presented in Section IV-B5, wherein the ProVerif verifier judged that our MD construction formalization with MD strengthening prevented the LEA. The formalization to represent the padding used the model in Section V-C. Please see Section V-C for details. We describe herein the kind of value that was actually formalized for padding.

The method formalized a block of bitstring terms comprising four oct terms. Thus, the final block of the split message bitstring term may leave zero, one, two, or three oct terms. If the number of the remaining oct terms in the final block is less than four, we fill the constant oct term "zero." Figure 9 illustrates padding with the oct term "zero" in the final block. In line 112 of Code.7, the first part of the final block was a message formalized with the assumption that the oct type constant term `zero` was added as a padding to the remaining three parts.

The padding process in this case was formalized as follows:

"`let p = con4octs(b2o(rm),zero,zero, zero).`" [6]

The verification result by ProVerif output "true," indicating that no successful length extension attack paths were found.

```
  (* Code.7: Formalization of MD-strengthening *)
1   free c:channel.
2   free s:channel[private].
3   free t:channel[private].
4   free u:channel[private].
5   const iv:bitstring.
6
7   fun con(bitstring,bitstring):bitstring.
8   fun divhead(bitstring):bitstring.
9   fun divrest(bitstring):bitstring.
10  equation forall mt:bitstring;
11   con(divhead(mt),divrest(mt))=mt.
12
13  (* Compress function *)
14  fun comp(bitstring,bitstring):bitstring.
15
16  fun length(bitstring):bitstring.
17
18  table MD_h(bitstring,bitstring).
19  table MD_h_len(bitstring,bitstring).
20  table MD_h_pad(bitstring,bitstring).
21  table MD_h_check(bitstring).
22
23  (* Padding *)
24  type oct.
25  const zero:oct.
26
27  fun b2o(bitstring):oct.
28  fun o2b(oct):bitstring.
29  equation forall xb:bitstring;o2b(b2o(xb))=xb.
30  equation forall xo:oct;b2o(o2b(xo))=xo.
31
32  fun Ocon(oct,oct):oct.
33  fun Odivhead(oct):oct.
34  fun Odivrest(oct):oct.
35  equation forall mt:oct;
36    Ocon(Odivhead(mt),Odivrest(mt))=mt.
37  equation forall lmt:oct,rmt:oct;
38    Odivhead(Ocon(lmt,rmt))=lmt.
39  equation forall lmt:oct,rmt:oct;
40    Odivrest(Ocon(lmt,rmt))=rmt.
41
42  fun oct1(bitstring):oct.
43  fun oct2i2(bitstring):oct.
44  fun oct2i34(bitstring):oct.
45  fun oct3i3(bitstring):oct.
46  fun oct3i4(bitstring):oct.
47  fun oct4(bitstring):oct.
48  fun con4octs(oct,oct,oct,oct):bitstring.
49
50  equation forall x:oct,y:oct,z:oct,w:oct;
51    con4octs(x,y,z,w)=
52      o2b(Ocon(x,Ocon(y,Ocon(z,w)))).
53
54  equation forall x:bitstring;
55    oct1(x)=Odivhead(b2o(x)).
56  equation forall x:bitstring;
57    oct2i2(x)=Odivrest(b2o(x)).
58  equation forall x:bitstring;
59    oct2i34(x)=Odivhead(Odivrest(b2o(x))).
60  equation forall x:bitstring;
61    oct3i3(x)=Odivrest(Odivrest(b2o(x))).
62  equation forall x:bitstring;
63    oct3i4(x)=
64      Odivhead(Odivrest(Odivrest(b2o(x)))).
65  equation forall x:bitstring;
66    oct4(x)=
67      Odivrest(Odivrest(Odivrest(b2o(x)))).
68
69  (* Query *)
70  event SUCCESS.
71  query event(SUCCESS).
72
```

[6]Similarly, when adding the oct type constant term zero as a padding to the remaining two parts or one, we formalized the padding process as "let p = con4octs(oct1(rm),oct2i2(rm),zero,zero)." or "let p = con4octs(oct1(rm),oct2i34(rm),oct3i3(rm),zero)."

```
73 (* MD construction *)
74 let makeMD =
75 in(s,(ha:bitstring,rm:bitstring,bl:nat, mm:bitstring));
76    if(bl <> 1) then
77    (
78      let newbl = divhead(rm) in
79      let newstream = divrest(rm) in
80      let newha = comp(ha,newbl) in
81      out(s,(newha,newstream,bl-1, mm))
82    ) else (
83      let MDh = comp(ha,rm) in
84      insert MD_h(mm, MDh)
85    ).
86
87  let makeMDlen =
88   in(t,(ha:bitstring,rm:bitstring,bl:nat,len:bitstring,
89       mm:bitstring));
90    if(bl <> 1) then
91    (
92      let newbl = divhead(rm) in
93      let newstream = divrest(rm) in
94      let newha = comp(ha,newbl) in
95      out(t,(newha,newstream,bl-1,len, mm))
96    ) else (
97      let newha = comp(ha,rm) in
98      let MDh = comp(newha,len) in
99      insert MD_h_len(mm, MDh)
100   ).
101
102 let makeMDpad =
103  in(u,(ha:bitstring,rm:bitstring,bl:nat,len:bitstring,
104      mm:bitstring));
105   if(bl <> 1) then
106   (
107     let newbl = divhead(rm) in
108     let newstream = divrest(rm) in
109     let newha = comp(ha,newbl) in
110     out(u,(newha,newstream,bl-1,len, mm))
111   ) else (
112     let p = con4octs(b2o(rm),zero,zero,zero) in
113     let newha = comp(ha,p) in
114     let MDh = comp(newha,len) in
115     insert MD_h_pad(mm, MDh)
116   ).
117
118 let LEA_CHECK =
119   in(c, MDh12':bitstring);
120   get MD_h_check(=MDh12') in
121   event SUCCESS.
122
123 (* Main process *)
124 process
125   (
126     new m1:bitstring;
127     new m2:bitstring;
128     out(c, (m2,2));
129     out(t,(iv,m1,2,length(m1), m1));
130     get MD_h_len(=m1, MDh1':bitstring) in
131     out(c,MDh1');
132     out(s,(iv,m1,2, m1));
133     get MD_h(=m1, MDh1:bitstring) in
134     out(u,(MDh1,m2,2,length(con(m1,m2)), con(m1,m2)));
135     get MD_h_pad(=con(m1,m2), MDh12:bitstring) in
136     insert MD_h_check(MDh12)
137   ) | !makeMD | !makeMDlen | !makeMDpad | !LEA_CHECK
```

V. FORMALIZATION OF THE CIPHER BLOCK CHAINING MODE IN PROVERIF

A. CIPHER BLOCK CHAINING MODE

This section presents a brief review of the CBC mode, which is a confidentiality mode with an encryption process that features a combination of plain blocks and the previous cipher blocks [51]. The CBC mode requires an *IV* to combine with the first plain block. Note that the *IV* does not need to be kept a secret, but must be unpredictable. Figure 10 illustrates the CBC mode.

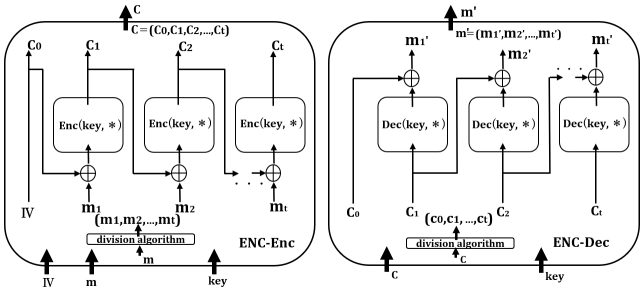

FIGURE 10. CBC Mode.

The CBC mode is generally defined as follows:
CBC encryption:

$$C_0 = IV \tag{5}$$

$$C_1 = Enc(key, m_1 \oplus IV) \tag{6}$$

$$C_j = Enc(key, m_j \oplus C_{j-1}) \quad for \quad j \leqq t. \tag{7}$$

CBC decryption:

$$m'_1 = Dec(key, C_1) \oplus IV \tag{8}$$

$$m'_j = Dec(key, C_j) \oplus C_{j-1} \quad for \quad j \leqq t. \tag{9}$$

The first input block of the CBC encryption was formed using an XOR operation, the first block of the plain block with the *IV*. The *Enc* function was applied to the first input block. The resulting output block was the first block of the cipher block. This output block was also obtained by performing an XOR operation with the second plain block to produce the second input block. The *Enc* function was employed to produce the second output block. This output block was the second cipher block obtained by performing an XOR operation with the next plain block to form the next input block. Each successive plain block was obtained by performing an XOR operation with the previous output cipher block to produce the new input block. The *Enc* function was applied to each input block to produce the cipher block.

The CBC decryption in the *Dec* function was applied to the first cipher block. The resulting output block was obtained by performing an XOR operation with the initialization vector to recover the first plain block. The *Dec* function was also applied to the second cipher block. The resulting output block was obtained by performing an XOR operation with the first cipher block to recover the second plain block. The *Dec* function must be applied to the corresponding cipher block to recover any plain block, except for the first plain block. The resulting block was obtained by performing an XOR operation with the previous cipher block.

In the CBC encryption, the input block to each *Enc* operation, except for the first, depends on the results of the previous *Enc* operation; thus, *Enc* operations cannot be performed in parallel. However, the CBC decryption denotes the input blocks for the *Dec* function, and the cipher blocks are immediately available; hence, multiple *Dec* operations can be performed in parallel.

B. FORMALIZATION OF THE CBC MODE CODE

We proposed herein the CBC mode formalization in ProVerif. For simplicity, we show the model formalizing the CBC mode without padding. Please refer to Section V-C for the CBC mode formalization with padding. Code.8 depicts the CBC mode formalization without padding.

We confirmed a successful encryption and decryption processing in the CBC mode and the secrecy of the CBC mode in our formalization using the ProVerif verifier. We present below the model formalizing the CBC mode without padding:

```
  (* Code.8: CBC mode *)
1   free d:channel[private].
2   free dd:channel[private].
3   free ee:channel[private].
4   free cc:channel[private].
5   free pubc:channel.
6   free ts:bitstring[private].
7   free sskey:bitstring[private].
8
9   (* concatenation and division of bitstring *)
10  fun con(bitstring,bitstring):bitstring.
11  fun divhead(bitstring):bitstring.
12  fun divrest(bitstring):bitstring.
13  equation forall mt:bitstring;
14   con(divhead(mt),divrest(mt))=mt.
15
16  equation forall lmt:bitstring,rmt:bitstring;
17   divhead(con(lmt,rmt))=lmt.
18
19  equation forall lmt:bitstring,rmt:bitstring;
20   divrest(con(lmt,rmt))=rmt.
21
22  (* XOR *)
23  const zeros: bitstring.
24  fun xor(bitstring,bitstring):bitstring.
25  equation forall x:bitstring,y:bitstring;
26   xor(xor(x,y),y) = x.
27  equation forall x:bitstring; xor(x,x) = zeros.
28  equation forall x:bitstring; xor(zeros,x) = x.
29  equation forall x:bitstring; xor(x,zeros) = x.
30
31  (*Block cipher*)
32  fun enc(bitstring,bitstring):bitstring.
33  fun dec(bitstring,bitstring):bitstring.
34  equation forall x: bitstring,
35  s: bitstring; dec(enc(x,s),s) = x.
36
37  event SuccCBC.
38
39  let CBCe(prskey:bitstring) =
40    in(d,(iv:bitstring, m:bitstring, n:nat));
41    if(n <> 1)then
42    (
43      let tb = divhead(m) in
44      let nb = divrest(m) in
45      let cipb = enc(xor(tb,iv),prskey)in
46      out(cc, (iv,cipb));
47      out(pubc, (iv,cipb)); (* for adversary *)
48      out(d,(cipb,nb,n-1))
49    )
50    else
51    (
52      let cipb = enc(xor(m,iv),prskey) in
53      out(pubc, (iv,cipb)); (* for adversary *)
54      out(cc, (iv,cipb))
55    ).
56
57  let CBCd(prskey:bitstring) =
58    in(dd, x2:nat);
59    in(cc, (ciphb1:bitstring,ciphb2:bitstring));
60    let plainb = dec(ciphb2,prskey) in
61    if (ciphb2 = enc(plainb,prskey)) then
62    let plaina = xor(plainb,ciphb1) in
63    if (plainb = xor(plaina,ciphb1)) then
64    let x3:nat = x2-1 in
65    if (x3 <> 0) then
66    (
67      out(dd, x3);
68      in(ee,(x4:nat,tailce:bitstring));
69      if(x4=x3) then
70      (
71        let conb = con(plaina,tailce) in
72        out(ee,(x2,conb));
73        if(conb=ts) then event SuccCBC
74      )
75    )
76    else
77      out(ee, (x2,plaina)).
78
79  (* queries *)
80  query attacker(ts).
81  query event(SuccCBC).
82
83  (* execution part *)
84  process
85    (
86      new iv:bitstring;
87      out(cc,(iv,4));
88      out(d,(iv,ts,4));
89      !(
90        out(dd, 4)
91       )
92    )
93    | !CBCe(sskey) | !CBCd(sskey)
```

In the first part of Code.8, we declared the terms and functions and defined the rewriting rules for the terms. In Code.8, the first part up to line 7 involved the declaration of terms, such as communication channel, plaintext, and secret key. The formal definitions of the functions for the concatenation and division of bitstrings, XOR operation, and block cipher are presented in lines 10–35 of Code.8. Line 37 shows the previous declaration of the event labeled as "`SuccCBC`." We then defined the iterative calculation process of the CBC mode.

We formalized the CBC encryption process as "`CBCe`" in lines 39–55 and decryption process as "`CBCd`" in lines 57–77. Lines 80 and 81 depict queries that tell ProVerif what to verify. Finally, we defined the protocol execution procedure to be verified in lines 84–93. Here, message block division was performed in the manner described in Section IV-B1. The "`con`" function (line 10 of Code.8) generated the concatenation of two bitstring type terms. The "`divhead`" and "`divrest`" functions (lines 11 and 12 of Code.8) extracted a bitstring portion. These functions took `divhead`, which extracted the first block of the given bitstring, and `divrest`, which extracted the portion other than that extracted by `divhead`, as arguments. Consequently, the bitstring term `X` was divided into bitstrings `divhead(X)` and `divrest(X)`, where `con(divhead(X),divrest(X)) = X`. In this model, we treated the lengths of bitstrings `X`, `divhead(X)`, and `divrest(X)` as "$\ell, 1, \ell - 1$." The block length of the two bitstrings must be kept as individual nat terms after division in the process because both `divhead` and `divrest` generated a bitstring type free term, as previously mentioned. We defined the rewriting rules for the concatenation and division of bitstrings in lines 13–20 of Code.8.

1) ITERATIVE EXECUTIONS OF THE CBC MODE

Section IV-B1 described the method for formalizing the bitstring term division. To formalize the CBC encryption

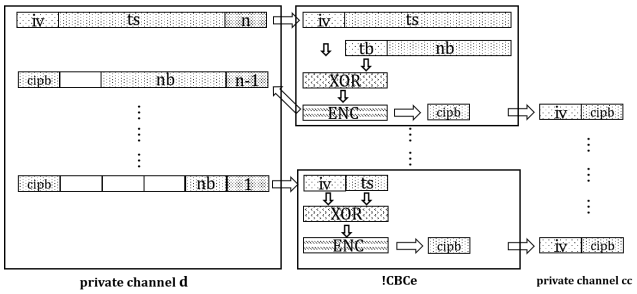

FIGURE 11. Formalization of the CBC encryption.

and decryption algorithm, we present herein the formal definitions of a model that iteratively executes the block division and encryption/decryption functions (Figure 11).

2) CBC ENCRYPTION:

The `CBCe` process in lines 39–55 of Code.8 depicts a formal model representing one step of the iterative execution of the CBC encryption. On the left side of line 93 of Code.8 in the execution part, "`!CBCe(sskey)`" refers to the iterative subroutine call as the execution of an unbounded number of `CBCe` process replications. "`sskey`" is an argument representing the shared secret key that each process `CBCe` replication takes at runtime.

First, the main process declares a local bitstring term *IV* for the CBC encryption. Next, the main process invokes `CBCe` as a subroutine by sending two bitstrings and a nat term via a private channel `d` (line 88 of Code.8) as follows: "`out(d,(iv,ts,4))`," where the terms in the triplet `(iv,ts,4)` are the initialization vector, plaintext `(ts)`, and plaintext length, respectively.

In ProVerif, we can declare a channel with the "private" attribute. Any term sent via a private channel is kept from the attacker. In the proposed model, we treat a private channel as an internal communication between the local processes. A `CBCe` replication process receives a triplet of terms `(iv,m,n)` via a secret channel `d` in line 40 of Code.8.

Here, `iv` is the cipher block in the previous step of `CBCe`; `m` is the remaining bitstring in the plaintext; and `n` is the `m` length. If `n` is not `1`, then the current step of `CBCe` divides `m` into the two bitstring terms of `tb = divhead(m)` and `nb = divrest(m)`, as shown in lines 43 and 44 of Code.8. In line 45, `CBCe` computes "`cipb = enc(xor(tb,iv),sskey)`" as the cipher block of this step. Note that `sskey = prskey`. `CBCe` then sends a pair of bitstrings `(iv,cipb)` to `CBCd` via the private channel `cc` in line 46. The previous ciphertext and the ciphertext pair `(iv,cipb)` are sent to control the cipher sequence order. These ciphertexts must be sent via the public channel such that attackers can obtain them; however, `cc` is a private channel. Due to a ProVerif specification issue, if `cc` is a public channel, then `CBCd` cannot perform a correct decryption. These ciphertexts are output to the public channel `pubc` in line 47 such that they can be obtained attackers. The current `CBCe` then sends `(cipb,nb,n-1)` to the next step of `CBCe` via the private channel `d` in line 48. Note that `n-1` is the length of `nb`. If `n = 1`, the current step is the final step of executing `CBCe`. `CBCe` then sends a pair of bitstrings `(iv,cipb)` to `CBCd` via the private channel `cc` in line 54.

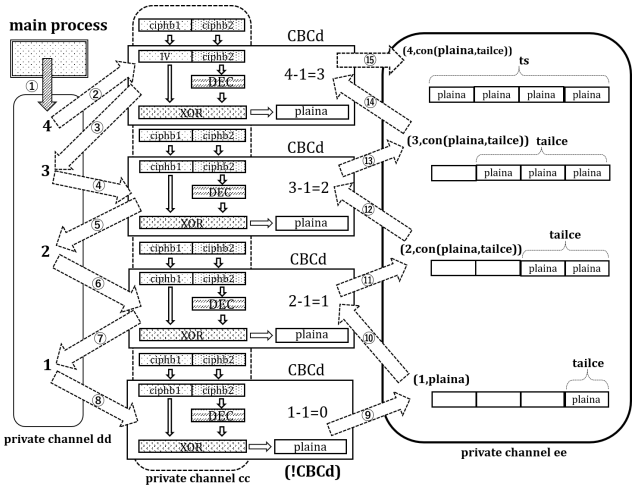

FIGURE 12. Formalization of the CBC decryption.

3) CBC DECRYPTION:

The `CBCd` process in lines 57–77 of Code.8 is a formal model representing one step of the iterative execution of the CBC decryption. On the right-hand side of line 93 in Code.8 in the execution part, "`!CBCd(sskey)`" represents an iterative subroutine call executing an unbounded number of the `CBCd` process replications. `CBCd` successively decrypted the ciphertext blocks received from `CBCe` via a private channel `cc` from the final block and concatenated the decrypted blocks to calculate the original plaintext. It must be formalized, such that other receiver processes can receive the ciphertext at once and call the CBC decryption process to prepare for the CBC mode usage. We, however, formalized the CBC decryption according to the description to simplify the model explanation process (Figure 12).

First, the main process invokes `CBCd` as a subroutine by sending the nat term representing the number of replications via the private channel `dd` in line 90 of Code.8 as follows: "`out(dd,4)`," where nat term 4 is the plaintext length `(ts)`. The `CBCd` replication processes communicate with each other via private channels `dd` and `ee`. `dd` is the channel for the counter of replications. `ee` is the channel for temporary data during an iterative execution. In other words, `ee` represents the internal shared memory for the decryption process, that is, `CBCd`.

The `CBCd` replication process receives a nat term `x2` via the `dd` in line 58 of Code.8. `CBCd` then receives a pair of bitstrings `(ciphb1,ciphb2)` via the private channel `cc` in line 59. This bitstring pair comprises the previous and current ciphertexts. `CBCd` computes "`plaina = xor(plainb,ciphb1)`" as the

plaintext of this step in lines 60–63. Here, "`plainb = dec(ciphb2,sskey)`." Note that the "`if`" statements in lines 61 and 63 are essentially unnecessary operations. Hence, if these "`if`" statements do not exist, and a private channel is used, the DEC and XOR function equations are not correctly applied due to a ProVerif specification. `CBCd` then computes `x3 = x2-1` in line 64.

If `x3 = 0`, `CBCd` sends `(x2,plaina)` as a return to the previous step of `CBCd` via `ee`. If `x3 ≠ 0`, `CBCd` sends `x3` to another replication of `CBCd` via `dd` for an iterative execution. The current `CBCd` then receives a pair of terms `(x4,tailce)` via a private channel `ee` from another `CBCd` replication. Note that the current `CBCd` must ensure that the received data come from the next step of `CBCd` considering the formal model characteristics of channels (Section II-C). `tailce` denotes a part of the plaintext already decrypted and concatenated. If `x4 = x3`, `(x4,tailce)` represents return from the next step. The current `CBCd` then returns `(x2,con(plaina,tailce))` to the previous step of `CBCd` or the main process via channel `ee`. Consequently, the `CBCd` process can decrypt the cipher block of a specified number of times via an iterative execution. Note that line 73 in Code.8 is for the query to confirm if this code works well or not; hence, it is not relevant in the iteration behavior formalization.

4) VERIFICATION RESULT

We successfully formalized the CBC mode behavior in ProVerif under execution conditions. We ensured that the abovementioned model functions as intended by verifying the reachability of the event `SuccCBC` defined in line 73 of Code.8, wherein encryption and decryption are completed by ProVerif. The ProVerif verifier found an execution path from the initiation of the execution to the event `SuccCBC`. Thus, once the main process invoked the first step of `CBCe`, each `CBCe` process replication started to communicate with the next via a private channel.

Consequently, the iterative execution of `CBCe` successfully computed the CBC encryption. We then verified whether the attacker can obtain the original message from the cipher blocks output to the public communication channel by the query "`query attacker(ts)`" in line 80 of Code.8. The ProVerif verification showed that "not attacker(ts) is true" was output, indicating that no successful attack path that breaks the CBC encryption secrecy was detected.

The padding oracle attack POODLE (CVE-2014-3566) [44], [45] sequentially restores the plaintext by performing an exhaustive search when the application protocol using the CBC mode contains a defect acting as a padding oracle. We extended a method for formalizing the iterative executions in ProVerif to handle the padding. This method explicitly defines the padding oracle model as a process. We reproduced the attack path by adding it the abovementioned verification code.

Section VI-C presents a detailed explanation for this.

5) CHOSEN-PLAINTEXT ATTACK

The CBC mode without a tampering check is weak against the CPA. We demonstrate herein the CPA behavior in our CBC mode formalization (Code.9).

In our CBC mode formalization, a legitimate sender calls the encryption process via a private channel. In this formalization, we allowed the attacker to encrypt any plaintext by introducing an encryption oracle process, which transfers the input plaintext from the public channel to the private one. The encryption oracle process was nearly the same as the main process of the legitimate sender, except for the input from the public channel shown in lines 39–47 of Code.9.

```
  (* Code.9: Formalization of Chosen-plaintext attack. *)
1   free d:channel[private].
2   free dd:channel[private].
3   free ee:channel[private].
4   free cc:channel[private].
5   free pubc:channel.
6   free ts:bitstring[private].
7   free sskey:bitstring[private].
8
9   (* concatenation and division of bitstring *)
10  fun con(bitstring,bitstring):bitstring.
11  fun divhead(bitstring):bitstring.
12  fun divrest(bitstring):bitstring.
13  equation forall mt:bitstring;
14   con(divhead(mt),divrest(mt))=mt.
15
16  equation forall lmt:bitstring,rmt:bitstring;
17   divhead(con(lmt,rmt))=lmt.
18  equation forall lmt:bitstring,rmt:bitstring;
19   divrest(con(lmt,rmt))=rmt.
20
21  (* XOR *)
22  const zeros: bitstring.
23  fun xor(bitstring,bitstring):bitstring.
24  equation forall x:bitstring,y:bitstring;
25   xor(xor(x,y),y) = x.
26
27  equation forall x:bitstring; xor(x,x) = zeros.
28  equation forall x:bitstring; xor(zeros,x) = x.
29  equation forall x:bitstring; xor(x,zeros) = x.
30
31  (*Block cipher*)
32  fun enc(bitstring,bitstring):bitstring.
33  fun dec(bitstring,bitstring):bitstring.
34  equation forall x: bitstring,
35  s: bitstring; dec(enc(x,s),s) = x.
36
37  event SuccCBC.
38
39  let CPAO(n:nat) =
40    (
41      in(pubc,(iv:bitstring, m:bitstring));
42      out(cc,(iv,n));
43      out(d,(iv,m,n));
44      !(
45      out(dd, n)
46      )
47    ).
48
49  let CBCe(prskey:bitstring) =
50    in(d,(iv:bitstring, m:bitstring, n:nat));
51    if(n <> 1)then
52    (
53      let tb = divhead(m) in
54      let nb = divrest(m) in
55      let cipb = enc(xor(tb,iv),prskey)in
56      out(cc, (iv,cipb));
57      out(pubc, (iv,cipb)); (* for adversary *)
58      out(d,(cipb,nb,n-1))
59    )
60    else
61    (
62      let cipb = enc(xor(m,iv),prskey) in
63      out(pubc, (iv,cipb)); (* for adversary *)
```

```
64      out(cc, (iv,cipb))
65    ).
66
67  let CBCd(prskey:bitstring) =
68    in(dd, x2:nat);
69    in(cc, (ciphb1:bitstring,ciphb2:bitstring));
70    let plainb = dec(ciphb2,prskey) in
71    if (ciphb2 = enc(plainb,prskey)) then
72    let plaina = xor(plainb,ciphb1) in
73    if (plainb = xor(plaina,ciphb1)) then
74    let x3:nat = x2-1 in
75    if (x3 <> 0) then
76    (
77      out(dd, x3);
78      in(ee,(x4:nat,tiv:bitstring,tailce:bitstring));
79      if(x4 = x3 && tiv = ciphb2) then
80      (
81        let conb = con(plaina,tailce) in
82        out(ee,(x2,ciphb1,conb));
83        in(pubc,X:bitstring);
84        if(conb = con(X,ts)) then event SuccCBC
85      )
86    )
87    else
88      out(ee, (x2,ciphb1,plaina)).
89
90  (* queries *)
91  query event(SuccCBC).
92
93  (* execution part *)
94  process
95    (
96      new iv:bitstring;
97      let tb = divhead(ts) in
98      let nb = divrest(ts) in
99      let civ= enc(iv,sskey)in
100    out(pubc,iv);
101    out(cc,(civ,2));
102    out(d,(civ,ts,2));
103    !(
104      out(dd, 2)
105    )
106   )
107   | !CBCe(sskey) | !CBCd(sskey) | !CPAO(3)
```

Lines 83–84 of Code.9 defined the following event in the decryption process to confirm the CPA attack. Here, `ts` is a private plaintext, and `X` is an attacker-specified bitstring.

In this case, the attacker receives the `iv` and the ciphertext of `ts` encrypted by the legitimate sender. If this event occurs, the ciphertext extension attack was successful using the chosen plaintext attack. Appendix B shows the verification results of the CPA.

C. CBC MODE FORMALIZATION WITH PADDING

This section introduces a method for handling padding in our formalization. We present herein a simple concept: only rewriting rules are added to handle padding in the final block of the process in the proposed method (Section V-B). In other words, the final block padding must be explicitly formalized. First, the unit length bitstring term was divided into a fixed number of parts. We declared the term type representing this part of the divided bitstrings as "`oct`." The following example in Code.10 was formalized, such that a bitstring term unit comprised four oct terms:

```
  (* Code.10:CBC mode with padding *)
1   free d:channel[private].
2   free dd:channel[private].
3   free ee:channel[private].
4   free cc:channel[private].
5   free pubc:channel.
6   free ts:bitstring[private].
7   free sskey:bitstring[private].
8
9   (* concatenation and division of bitstring *)
10  fun con(bitstring,bitstring):bitstring.
11  fun divhead(bitstring):bitstring.
12  fun divrest(bitstring):bitstring.
13  equation forall mt:bitstring;
14   con(divhead(mt),divrest(mt))=mt.
15
16  equation forall lmt:bitstring,rmt:bitstring;
17   divhead(con(lmt,rmt))=lmt.
18
19  equation forall lmt:bitstring,rmt:bitstring;
20   divrest(con(lmt,rmt))=rmt.
21
22  (* XOR *)
23  const zeros: bitstring.
24  fun xor(bitstring,bitstring):bitstring.
25  equation forall x:bitstring,y:bitstring;
26   xor(xor(x,y),y) = x.
27  equation forall x:bitstring; xor(x,x) = zeros.
28  equation forall x:bitstring; xor(zeros,x) = x.
29  equation forall x:bitstring; xor(x,zeros) = x.
30
31  (* Block cipher *)
32  fun enc(bitstring,bitstring):bitstring.
33  fun dec(bitstring,bitstring):bitstring.
34  equation forall x: bitstring, s: bitstring;
35   dec(enc(x,s),s) = x.
36
37  (* func for Oct *)
38  type oct.
39  const CPAD:oct.
40  const CPAD1:oct.
41  const CPAD2:oct.
42  const CPAD3:oct.
43  const CPAD4:oct.
44
45  fun b2o(bitstring):oct.
46  fun o2b(oct):bitstring.
47  equation forall xb:bitstring;o2b(b2o(xb))=xb.
48  equation forall xo:oct;b2o(o2b(xo))=xo.
49
50  fun Ocon(oct,oct):oct.
51  fun Odivhead(oct):oct.
52  fun Odivrest(oct):oct.
53  equation forall mt:oct; Ocon(Odivhead(mt),
54   Odivrest(mt))=mt.
55  equation forall lmt:oct,rmt:oct;
56   Odivhead(Ocon(lmt,rmt))=lmt.
57  equation forall lmt:oct,rmt:oct;
58   Odivrest(Ocon(lmt,rmt))=rmt.
59
60  fun oct1(bitstring):oct.
61  fun oct2i2(bitstring):oct.
62  fun oct2i34(bitstring):oct.
63  fun oct3i3(bitstring):oct.
64  fun oct3i4(bitstring):oct.
65  fun oct4(bitstring):oct.
66  fun con4octs2(oct,oct,oct,oct):bitstring.
67
68  equation forall x:oct,y:oct,z:oct,w:oct;
69  con4octs2(x,y,z,w)=o2b(Ocon(x,Ocon(y,Ocon(z,w)))).
70
71  equation forall x:bitstring;
72   oct1(x)=Odivhead(b2o(x)).
73  equation forall x:bitstring;
74   oct2i2(x)=Odivrest(b2o(x)).
75  equation forall x:bitstring;
76   oct2i34(x)=Odivhead(Odivrest(b2o(x))).
77  equation forall x:bitstring;
78   oct3i3(x)=Odivrest(Odivrest(b2o(x))).
79  equation forall x:bitstring;
80   oct3i4(x)=Odivhead(Odivrest(Odivrest(b2o(x)))).
81  equation forall x:bitstring;
82   oct4(x)=Odivrest(Odivrest(Odivrest(b2o(x)))).
83
84  event SuccCBC.
85
86  let CBCe(prskey:bitstring) =
87    in(d,(iv:bitstring, m:bitstring, n:nat));
88    if(n <> 1)then
89    (
```

```
90      let tb = divhead(m) in
91      let nb = divrest(m) in
92      let cipb = enc(xor(tb,iv),prskey)in
93      out(cc, (iv,cipb));
94      out(pubc, (iv,cipb)); (* for adversary *)
95      out(d,(cipb,nb,n-1))
96    )
97    else
98    (
99      (* for 3 octets padding *)
100     let p = con4octs2(b2o(m),CPAD3,CPAD3,CPAD3) in
101     let cipb = enc(xor(p,iv),prskey) in
102     out(pubc, (iv,cipb)); (* for adversary *)
103     out(cc, (iv,cipb))
104   ).
105 106 let CBCd(prskey:bitstring) =
107   in(dd, x2:nat);
108   in(cc, (ciphb1:bitstring,ciphb2:bitstring));
109   let plainb = dec(ciphb2,prskey) in
110   if (ciphb2 = enc(plainb,prskey)) then
111   let plaina = xor(plainb,ciphb1) in
112   if (plainb = xor(plaina,ciphb1)) then
113   let x3:nat = x2-1 in
114   if (x3 <> 0) then
115   (
116     out(dd, x3);
117     in(ee,(x4:nat,tailce:bitstring));
118     if(x4=x3) then
119     (
120       let conb = con(plaina,tailce) in
121       out(ee,(x2,conb));
122       if(conb=ts) then event SuccCBC
123     )
124   )
125   else
126   (
127     (* for 3 octets padding *)
128     if (oct4(plaina) = CPAD3) then
129     if (oct3i4(plaina) = CPAD3) then
130     if (oct2i34(plaina) = CPAD3) then
131     let planac = oct1(plaina) in
132     out(ee, (x2,o2b(planac)))
133   ).
134
135 (* Queries *) 136 query attacker(ts). 137 query
    event(SuccCBC).
138
139 (* Execution part *) 140 process
141   (
142     new iv:bitstring;
143     out(cc,(iv,4));
144     out(d,(iv,ts,4));
145     !(
146       out(dd, 4)
147     )
148   )
149   | !CBCe(sskey) | !CBCd(sskey)
```

Lines 45–48 of Code.10 defined the mutual conversion functions of bitstring and oct as `b2o` and `o2b`, respectively. We assumed herein that the bitstring length of the argument of the `b2o` function was the same as the oct length. A bitstring free term has no information other than its name and type (Section IV-B1).

In lines 50–58 of Code.10, we formalized the concatenation and division of octs following the manner by which those of the bitstring type were formalized (Section IV-B1). In lines 60–82 of Code.10, we formalized the function of the concatenation of four octs into a unit bitstring and the extraction of each oct term from a unit bitstring term. In lines 68 and 69 of Code.10, o1, o2, o3, and o4 were treated as the "`oct`" terms. Thus, `UB = con4octs2(o1,o2,o3,o4)` is the unit bitstring term that represents the concatenation of these terms "`o1‖o2‖o3‖o4`." Here, `oct1(UB)`, `oct2i34(UB)`, `oct3i4(UB)`, `oct4(UB)` are `o1,o2,o3,` and `o4`, are `o1` and `o2`, respectively.

We formalized the CBC mode with padding by focusing on the case wherein the last block of the plaintext was a single oct, and the remaining single octs were padded with a fixed value `CPAD3`. In this case, the padding process was formalized as follows:
"`let p = con4octs2(b2o(m),CPAD3,CPAD3,CPAD3).`" [7]

VI. DISCUSSION

A. OUR PROPOSAL METHOD

Dealing with loop iterations is a popular theme for formal verification. In this work, we proposed a method for formalizing iterative executions as a formal model of function calls. Our method formalized the function calls with a specific focus on the formalization of the iterative executions in ProVerif using S/KEY (III-A), MD construction (IV-B2) and CBC mode (V-B1,V-B2,V-B3) as examples. We also confirmed the validity of the proposed formalization through case studies on the Merkle–Damgård structures of hash functions and the CBC mode of block ciphers. Our proposed method was consistent with the formal MD construction model and satisfied the properties of the cryptographic hash functions, especially, the CR (Section IV-B3). We formalized the CBC mode with and without padding and verified that we can encrypt/decrypt a plaintext comprising multiple blocks in the CBC mode while keeping the communication in the CBC mode a secret. The proposed formalization can discover the vulnerabilities imposed by the structures of the hash functions and the encryption modes (e.g., LEA and CPA). In a formal verification, hash functions and block ciphers are formalized as ideal functions without vulnerabilities. Considering this, we tested the validity of our formalization and validation results by formalizing the LEA on iterated hash functions (Section IV-B4), MD-strengthening method that makes the attack impossible (Section IV-B6), and CPA on the CBC mode (Section V-B5).

The results show that our method enables the formalization of the iterative methods used in complex cryptographic protocols in ProVerif and can be applied to the design of other cryptographic algorithms. This approach allows the formalization and security verification of cryptographic modules. Our approach is formalized as iterative execution, recursive execution can also be formalized using our proposed method.

B. LIMITATION

The number of loop iterations must be explicitly specified. Hence, our proposed formalizations were slightly different

[7]In case the last block of the plaintext is split into two octs, and the remaining two octs are padded with a fixed value (CPAD2), we formalized the padding process as
"let p = con4octs2(oct1(m),oct2i2(m),CPAD2,CPAD2)."

from their actual implementation, during which the number of iterations varied with the input length. The cryptographic protocols for verification were abstract models; therefore, whether they were secure was not strictly known. Real-world vulnerabilities may be overlooked by the formal verification results.

A remaining open problem is the formalization of the behavior of a general padding oracle itself. In the future work, we will apply our research results to higher-order differential attacks [54].

The subsequent section VI-C demonstrates that ProVerif derives a padding oracle attack against the CBC mode. In the derivation, ProVerif derives the execution path of the padding oracle attack if a padding oracle exists. As a result of the ProVerif verification, an attacker succeeds in the cipher analysis against the CBC mode if a padding oracle exists. However, our research cannot cover the formalization of the behavior of the padding oracle itself, and formalization remains as an outstanding problem. Our proposed method can be used to find the defects of a cryptographic protocol, which are causing a padding oracle.

C. DERIVATION OF THE PADDING ORACLE ATTACK AGAINST THE CBC MODE

Code.11 depicts the formalization of the padding oracle attack. ProVerif derived a padding oracle attack [44], [45] against the CBC mode. The padding oracle attack is important.

In this derivation, ProVerif derives the execution path of the padding oracle attack if a padding oracle exists. In other words, in a ProVerif verification result, the attacker can perform a successful cipher analysis against the CBC mode if a padding oracle exists. We cannot, however, formalize the padding oracle behavior.

Our proposed method can be used to identify the defects of a cryptographic protocol that cause the padding oracle. The padding oracle attack, called POODLE (CVE-2014-3566), sequentially restores the plaintext by performing an exhaustive search when an application protocol using the CBC mode contains a defect acting as a padding oracle. A security hole that exists in the implementation of the CBC mode or the application protocol that uses the CBC mode is used as a padding oracle attack. In a practical CBC decryption, the decryption process returns the plaintext if the decrypted plaintext padding is valid. In general, the attacker can decrypt the message using the decryption server as the padding oracle by identifying the padding value and using the message returned by the vulnerable decryption server regarding the padding verification. However, to formalize, the padding oracle discovery requires some ingenuity.

```
  (* Code.11:CBC mode with padding oracle attack *)
1   free d:channel[private].
2   free dd:channel[private].
3   free ee:channel[private].
4   free cc:channel[private].
5   free pubc:channel.
6   free ts:bitstring[private].
7   free sskey:bitstring[private].
8
9   (* concatenation and division of bitstring *)
10  fun con(bitstring,bitstring):bitstring.
11  fun divhead(bitstring):bitstring.
12  fun divrest(bitstring):bitstring.
13  equation forall mt:bitstring;
14   con(divhead(mt),divrest(mt))=mt.
15
16  equation forall lmt:bitstring,rmt:bitstring;
17   divhead(con(lmt,rmt))=lmt.
18
19  equation forall lmt:bitstring,rmt:bitstring;
20   divrest(con(lmt,rmt))=rmt.
21
22  (* XOR *)
23  const zeros: bitstring.
24  fun xor(bitstring,bitstring):bitstring.
25  equation forall x:bitstring,y:bitstring;
26   xor(xor(x,y),y) = x.
27  equation forall x:bitstring; xor(x,x) = zeros.
28  equation forall x:bitstring; xor(zeros,x) = x.
29  equation forall x:bitstring; xor(x,zeros) = x.
30
31  (* Block cipher *)
32  fun enc(bitstring,bitstring):bitstring.
33  fun dec(bitstring,bitstring):bitstring.
34  equation forall x: bitstring, s: bitstring;
35   dec(enc(x,s),s) = x.
36
37  (* func for Oct *)
38  type oct.
39  const CPAD1:oct.
40  const CPAD2:oct.
41
42  fun b2o(bitstring):oct.
43  fun o2b(oct):bitstring.
44  equation forall xb:bitstring;o2b(b2o(xb))=xb.
45  equation forall xo:oct;b2o(o2b(xo))=xo.
46
47  fun Ocon(oct,oct):oct.
48  fun Odivhead(oct):oct.
49  fun Odivrest(oct):oct.
50  equation forall mt:oct;
51   Ocon(Odivhead(mt),Odivrest(mt))=mt.
52  equation forall lmt:oct,rmt:oct;
53   Odivhead(Ocon(lmt,rmt))=lmt.
54  equation forall lmt:oct,rmt:oct;
55   Odivrest(Ocon(lmt,rmt))=rmt.
56
57  fun oct1(bitstring):oct.
58  fun oct2i2(bitstring):oct.
59
60  fun con2octs(oct,oct):bitstring.
61  equation forall x:oct,y:oct;
62   con2octs(x,y)=o2b(Ocon(x,y)).
63
64  equation forall x:bitstring;
65   oct1(x)=Odivhead(b2o(x)).
66  equation forall x:bitstring;
67   oct2i2(x)=Odivrest(b2o(x)).
68
69  (* cheat code for attacker *)
70  fun AB1(bitstring):bitstring.
71  fun AB2(bitstring):bitstring.
72
73  fun Ccase1(bitstring,bitstring):bool
74  reduc forall Y:bitstring; Ccase1(AB1(Y),Y) = true
75  otherwise forall X:bitstring, Y:bitstring;
76  Ccase1(X,Y)= false.
77
78  fun Ccase2(bitstring,bitstring):bool
79  reduc forall Y:bitstring; Ccase2(AB2(Y),Y) = true
80  otherwise forall X:bitstring, Y:bitstring;
81  Ccase2(X,Y) = false.
82
83  event SuccCBC.
84
85  (* Adversary process *)
86  let processAdv(psskey:bitstring) =
87    in(pubc,(X:bitstring,Y:bitstring));
88    let py=dec(Y,psskey) in
89    let xpy =xor(X,py) in
90    (
91    if(Ccase1(X,Y)=true) then
92      out(pubc,oct1(py))
93    else
94    (
```

```
95    if(Ccase2(X,Y)=true) then
96      out(pubc,oct2i2(py))
97      )
98    ).
99 100~let CBCe(prskey:bitstring) =
101   in(d,(iv:bitstring, m:bitstring, n:nat));
102   if(n <> 1)then
103   (
104   let tb = divhead(m) in
105   let nb = divrest(m) in
106   let cipb = enc(xor(tb,iv),prskey)in
107   out(cc, (iv,cipb));
108   out(pubc, (iv,cipb)); (* for attacker *)
109   out(d,(cipb,nb,n-1))
110   )
111   else
112   (
113   (* for 1~octet padding *)
114   let p = con2octs(b2o(m),CPAD1) in
115   let cipb = enc(xor(p,iv),prskey) in
116   out(pubc, (iv,cipb)); (* for attacker *)
117   out(cc, (iv,cipb))
118   ).
119 120~let CBCd(prskey:bitstring) =
121   in(dd, x2:nat);
122   in(cc, (ciphb1:bitstring,ciphb2:bitstring));
123   let plainb = dec(ciphb2,prskey) in
124   if (ciphb2 = enc(plainb,prskey)) then
125   let plaina = xor(plainb,ciphb1) in
126   if (plainb = xor(plaina,ciphb1)) then
127   let x3:nat = x2-1 in
128   if (x3 <> 0) then
129   (
130     out(dd, x3);
131     in(ee,(x4:nat,tailce:bitstring));
132     if(x4=x3) then
133     (
134       let conb = con(plaina,tailce) in
135       out(ee,(x2,conb));
136       if(conb=ts) then event SuccCBC
137     )
138   )
139   else
140   (
141     (* for 1~octet padding *)
142     if (oct2i2(plaina) = CPAD1) then
143     let planac = oct1(plaina) in
144     out(ee, (x2,o2b(planac)))
145   ).
146
147 (* Queries *) 148~query attacker(ts). 149~query
    event(SuccCBC).
150
151 (* Execution part *) 152~process
153   (
154     new iv:bitstring;
155     out(cc,(iv,2));
156     out(d,(iv,ts,2));
157     !(
158       out(dd, 2)
159     )
160   )
161   | !CBCe(sskey) | !CBCd(sskey) | !processAdv(sskey)
```

Appendix C presents the verification result of the derivation of the padding oracle attack against the CBC mode.

VII. CONCLUSION AND FUTUREWORKS

In this study, we proposed a method for formalizing iterative executions in ProVerif. Our method formalizes function calls by treating them as communications between internal processes. We then formalized iterative execution as communication with self-duplicating processes. We specifically confirmed the validity of the proposed formalization through case studies on the MD construction algorithm, a method for constructing a cryptographic hash function using the proposed method. As verified, the proposed method was consistent with the formal MD construction model and satisfied the cryptographic hash function properties (i.e., CR). We also formalized the CBC mode, a well-known block cipher mode of operation. We verified that our formalization approach successfully described the encryption and decryption of a plaintext comprising multiple blocks in the CBC mode and their secrecy.

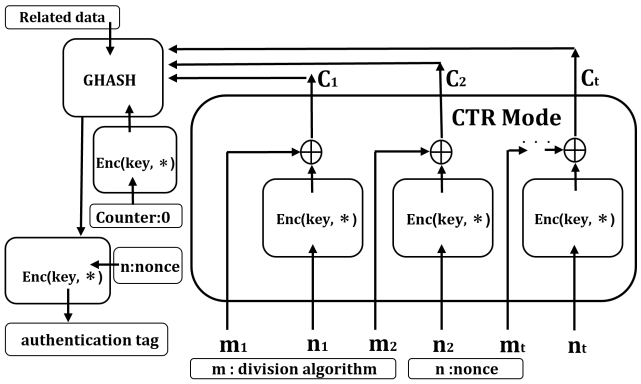

FIGURE 13. AES-GCM.

We aim to find security holes in the applied protocols [52] using ProVerif and help design cryptographic protocols. For example, our method can find the execution path of the padding oracle attack against our formal model of the CBC mode via automatic search if a padding oracle exists (Section VI-C). The proposed method allows the verification of the secure design of cryptographic algorithms and protocols using ProVerif. Thus, we attempt herein to formalize cryptographic protocols in a model closer to the actual implementation. That being said, we specifically attempt to formalize the complex application protocols (e.g., AES-Galois/Counter Mode (GCM)) (Figure 13) used. The GCM is an algorithm for authenticated encryption with associated data adopted in TLS 1.3. In addition, we plan to investigate applying the proposed method to aid cryptography, for example, in finding the properties of higher order differentials (e.g., saturation properties [53]) in higher order differential attacks [54].

APPENDIX A
VERIFICATION RESULT OF DERIVATION OF LEA

In the following, only the important log parts were extracted. The rest were omitted. See below for a log of the verification results on the derivation of the LEA in Code.5.

```
  (* Log:Length Extension Attack Log *)
(
    {1}new m1: bitstring;
    {2}new m2: bitstring;
    {3}out(c, (m2,2));
    {4}out(s, (iv,m1,2,m1));
    {9}get MD_h(=m1,MDh1: bitstring) in
    {5}out(c, MDh1);
    {6}out(s, (MDh1,m2,2,con(m1,m2)));
    {8}get MD_h(=con(m1,m2),MDh12: bitstring) in
    {7}insert MD_h_check(MDh12)
) | (
    {10}!
    {11}in(s, (ha: bitstring,rm: bitstring,bl: nat,mm: bitstring));
    {12}if (bl ≠ 1) then
    (
        {13}let newbl: bitstring = divhead(rm) in
        {14}let newstream: bitstring = divrest(rm) in
        {15}let newha: bitstring = comp(ha,newbl) in
        {16}out(s, (newha,newstream,bl - 1,mm))
    )
    else
        {17}let MDh: bitstring = comp(ha,rm) in
```

```
        {18}insert MD_h(mm,MDh)
) | (
    {19}!
    {20}in(c, MDhl2': bitstring);
    {22}get MD_h_check(=MDhl2') in
    {21}event SUCCESS
)

 Starting query not event(SUCCESS)
 goal reachable: event(SUCCESS)

 Derivation:

1. The message (m2[],2) may be sent to the attacker at output {3}.
  attacker((m2[],2)).
      :
  (omitted)
      :
8. The entry MD_h(ml[],comp(comp(iv,divhead(ml[])),divrest(ml[])))
  that may be in a table by 7~may be read at get {9}.
  So the message comp(comp(iv,divhead(ml[])),divrest(ml[])) may be
  sent to the attacker at output {5}.
  attacker(comp(comp(iv,divhead(ml[])),divrest(ml[]))).
      :
  (omitted)
      :
19. The message comp(comp(comp(comp(iv,divhead(ml[])),divrest(ml[])),
  divhead(m2[])),divrest(m2[])) that the attacker may have by 10
  may be received at input {20}.
  The entry MD_h_check(comp(comp(comp(comp(iv,divhead(ml[])),
  divrest(ml[])),divhead(m2[])),divrest(m2[]))) that may be in a
  table by 18~may be read at get {22}.
  So event SUCCESS may be executed at {21}.
  event(SUCCESS).
20. By 19, event(SUCCESS).
  The goal is reached, represented in the following fact:
  event(SUCCESS).
```

APPENDIX B
VERIFICATION RESULT OF THE CPA DERIVATION

In the following, only the important log parts were extracted. The rest were omitted. See below for a log of the verification results for the derivation of the chosen plaintext attack in Code.9.

```
  (* Log:Chosen Plaintext Attack Log *)
 (
    {1}new iv: bitstring;
    {2}let tb: bitstring = divhead(ts) in
    {3}let nb: bitstring = divrest(ts) in
    {4}let civ: bitstring = enc(iv,sskey) in
    {5}out(pubc, iv);
    {6}out(cc, (civ,2));
    {7}out(d, (civ,ts,2));
    {8}!
    {9}out(dd, 2)
 ) | (
    {10}!
    {11}let prskey: bitstring = sskey in
    {12}in(d, (iv_1: bitstring,m: bitstring,n: nat));
    {13}if (n ≠ 1) then
    (
        {14}let tb_1: bitstring = divhead(m) in
        {15}let nb_1: bitstring = divrest(m) in
        {16}let cipb: bitstring = enc(xor(tb_1,iv_1),prskey) in
        {17}out(cc, (iv_1,cipb));
        {18}out(pubc, (iv_1,cipb));
        {19}out(d, (cipb,nb_1,n - 1))
    )
    else
        {20}let cipb_1: bitstring = enc(xor(m,iv_1),prskey) in
        {21}out(pubc, (iv_1,cipb_1));
        {22}out(cc, (iv_1,cipb_1))
 ) | (
    {23}!
    {24}let prskey_1: bitstring = sskey in
    {25}in(dd, x2: nat);
    {26}in(cc, (ciphb1: bitstring,ciphb2: bitstring));
    {27}let plainb: bitstring = dec(ciphb2,prskey_1) in
    {28}if (ciphb2 = enc(plainb,prskey_1)) then
    {29}let plaina: bitstring = xor(plainb,ciphb1) in
    {30}if (plainb = xor(plaina,ciphb1)) then
    {31}let x3: nat = x2 - 1~in
    {32}if (x3 ≠ 0) then
    (
        {33}out(dd, x3);
        {34}in(ee, (x4: nat,tiv: bitstring,tailce: bitstring));
        {35}if ((x4 = x3) && (tiv = ciphb2)) then
        {36}let conb: bitstring = con(plaina,tailce) in
        {37}out(ee, (x2,ciphb1,conb));
        {38}in(pubc, X: bitstring);
        {39}if (conb = con(X,ts)) then
        {40}event SuccCBC
    )
    else
        {41}out(ee, (x2,ciphb1,plaina))
 ) | (
    {42}!
    {43}let n_1: nat = 3~in
    {44}in(pubc, (iv_2: bitstring,m_1: bitstring));
    {45}out(cc, (iv_2,n_1));
    {46}out(d, (iv_2,m_1,n_1));
    {47}!
    {48}out(dd, n_1)
 )

 Starting query not event(SuccCBC)
 goal reachable: event(SuccCBC)

 Derivation:

1. The attacker has some term m_2. attacker(m_2).
      :
  (omitted)
      :
10. The message (iv[],con(xor(iv[],iv[]),rmt)) that the attacker
  may have by 9~may be received at input {44}.
  So the message (iv[],con(xor(iv[],iv[]),rmt),3) may be sent on
  channel d[] at output {46}.
  mess(d[],(iv[],con(xor(iv[],iv[]),rmt),3)).
      :
  (omitted)
      :
22. Using the function zeros, the attacker may obtain zeros.
  attacker(zeros).
23. The message 3~that may be sent on channel dd[] by 4~may be
  received at input {25}.
  The message (iv[],enc(iv[],sskey[])) that may be sent on channel
  cc[] by 11~may be received at input {26}.
  The message (2,enc(iv[],sskey[]),ts[]) that may be sent on
  channel ee[] by 21~may be received at input {34}.
  The message zeros that the attacker may have by 22~may be
  received at input {38}.
  We have~2 ≠ 0.
  So event SuccCBC may be executed at {40}.
  event(SuccCBC).
24. By 23, event(SuccCBC).
  The goal is reached, represented in the following fact:
  event(SuccCBC).
```

APPENDIX C
VERIFICATION RESULT OF THE DERIVATION OF THE PADDING ORACLE ATTACK AGAINST THE CBC MODE

In the following, only the important log parts were extracted. The rest were omitted. See below for a log of the verification results on the derivation of the padding oracle attack against the CBC mode in Code.11.

```
(* Log:Padding Oracle Attack Log *)
 (
    {1}new iv: bitstring;
    {2}out(cc, (iv,2));
    {3}out(d, (iv,ts,2));
    {4}     {5}out(dd, 2)
) | (
    {6}     {7}let prskey: bitstring = sskey in
    {8}in(d, (iv_1: bitstring,m: bitstring,n: nat));
    {9}if (n ≠ 1) then
    (
        {10}let tb: bitstring = divhead(m) in
        {11}let nb: bitstring = divrest(m) in
        {12}let cipb: bitstring = enc(xor(tb,iv_1),prskey) in
        {13}out(cc, (iv_1,cipb));
        {14}out(pubc, (iv_1,cipb));
        {15}out(d, (cipb,nb,n - 1))
    )
    else
        {16}let p: bitstring = con2octs(b2o(m),CPAD1) in
        {17}let cipb_1: bitstring = enc(xor(p,iv_1),prskey) in
        {18}out(pubc, (iv_1,cipb_1));
        {19}out(cc, (iv_1,cipb_1))
) | (
    {20}     {21}let prskey_1: bitstring = sskey in
    {22}in(dd, x2: nat);
    {23}in(cc, (ciphb1: bitstring,ciphb2: bitstring));
    {24}let plainb: bitstring = dec(ciphb2,prskey_1) in
    {25}if (ciphb2 = enc(plainb,prskey_1)) then
    {26}let plaina: bitstring = xor(plainb,ciphb1) in
    {27}if (plainb = xor(plaina,ciphb1)) then
    {28}let x3: nat = x2 - 1~in
    {29}if (x3 ≠ 0) then
    (
        {30}out(dd, x3);
        {31}in(ee, (x4: nat,tailce: bitstring));
        {32}if (x4 = x3) then
        {33}let conb: bitstring = con(plaina,tailce) in
        {34}out(ee, (x2,conb));
        {35}if (conb = ts) then
        {36}event SuccCBC
    )
    else
        {37}if (oct2i2(plaina) = CPAD1) then
        {38}let planac: oct = oct1(plaina) in
        {39}out(ee, (x2,o2b(planac)))
) | (
    {40}     {41}let psskey: bitstring = sskey in
    {42}in(pubc, (X: bitstring,Y: bitstring));
    {43}let py: bitstring = dec(Y,psskey) in
    {44}let xpy: bitstring = xor(X,py) in
    {45}if (Ccase1(X,Y) = true) then
        {46}out(pubc, oct1(py))
    else
        {47}if (Ccase2(X,Y) = true) then
        {48}out(pubc, oct2i2(py))
 )

 Starting query not attacker(ts[])
```

```
goal reachable: attacker(ts[])

Derivation:

    :
(omitted)
    :
9. The message (AB2(enc(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])),sskey[])),
 enc(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])),sskey[]))
 that the attacker may have by 8~may be received at input {42}.
 So the message Odivrest(b2o(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])))
 may be sent to the attacker at output {48}.
 attacker(Odivrest(b2o(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])))))).
    :
 (omitted)
    :
12. The message (AB1(enc(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])),sskey[])),
 enc(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])),sskey[]))
 that the attacker may have by 11~may be received at input {42}.
 So the message Odivhead(b2o(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])))) may be
 sent to the attacker at output {46}.
 attacker(Odivhead(b2o(xor(o2b(Ocon(b2o(divrest(ts[])),CPAD1)),
 enc(xor(divhead(ts[]),iv[]),sskey[])))))).
    :
 (omitted)
    :
20. The message (AB2(enc(xor(divhead(ts[]),iv[]),sskey[])),
 enc(xor(divhead(ts[]),iv[]),sskey[]))
 that the attacker may have by 19~may be received at input {42}.
 So the message Odivrest(b2o(xor(divhead(ts[]),iv[]))) may be
 sent to the attacker at output {48}.
 attacker(Odivrest(b2o(xor(divhead(ts[]),iv[])))).
    :
 (omitted)
    :
23. The message (AB1(enc(xor(divhead(ts[]),iv[]),sskey[])),
 enc(xor(divhead(ts[]),iv[]),sskey[]))
 that the attacker may have by 22~may be received at input {42}.
 So the message Odivhead(b2o(xor(divhead(ts[]),iv[]))) may be
 sent to the attacker at output {46}.
 attacker(Odivhead(b2o(xor(divhead(ts[]),iv[])))).
24. By 23, the attacker may know Odivhead(b2o(xor(divhead(ts[]),iv[]))).
 By 20, the attacker may know Odivrest(b2o(xor(divhead(ts[]),iv[]))).
 Using the function con2octs the attacker may obtain
 xor(divhead(ts[]),iv[]).
 attacker(xor(divhead(ts[]),iv[])).
25. By 24, the attacker may know xor(divhead(ts[]),iv[]).
 By 17, the attacker may know iv[].
 Using the function xor the attacker may obtain divhead(ts[]).
 attacker(divhead(ts[])).
26. By 25, the attacker may know divhead(ts[]).
 By 16, the attacker may know divrest(ts[]).
 Using the function con the attacker may obtain ts[].
 attacker(ts[]).
    :
 (omitted)
    :
```

ACKNOWLEDGMENT

The authors wish to thank Enago for its linguistic assistance during the preparation of this paper.

REFERENCES

[1] B. Blanchet. *ProVerif: Cryptographic Protocol Verifier in the Formal Model*. Accessed: Jan. 2024. [Online]. Available: https://bblanche.gitlabpages.inria.fr/proverif/

[2] B. Blanchet, B. Smyth, and V. Cheval. *ProVerif 2.05: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial*. Accessed: Jan. 2024. [Online]. Available: https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf/

[3] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Found. Trends Privacy Secur.*, vol. 1, nos. 1–2, pp. 1–135, 2016, doi: 10.1561/3300000004.

[4] D. Basin, C. Cremers, J. Dreier, S. Meier, R. Sasse, and B. Schmidt. *Tamarin Prover*. Accessed: Jan. 2024. [Online]. Available: https://tamarin-prover.github.io/

[5] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie–Hellman protocols and advanced security properties," in *Proc. IEEE 25th Comput. Secur. Found. Symp.*, Jun. 2012, pp. 78–94.

[6] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov, "A meta-notation for protocol analysis," in *Proc. 12th IEEE Comput. Secur. Found. Workshop*, Jun. 1999, pp. 55–69.

[7] C. Cremers, *Scyther 1.1.3: Automatic Verification of Security Protocols, Scyther User Manual*. Accessed: Jan. 2024. [Online]. Available: https://github.com/cascremers/scyther/blob/master/gui/scyther-manual.pdf

[8] N. Kobeissi. *Verifpal*. Accessed: Jan. 2024. [Online]. Available: https://verifpal.com/

[9] A. Armando. (2005). *AVISPA 1.1: Automated Validation of Internet Security Protocols and Applications*. User Manual Tutorial. Accessed: Jan. 2024. [Online]. Available: https://www.avispa-project.org/

[10] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 2, pp. 198–208, Mar. 1983.

[11] R. Merkle, "One way hash functions and DES," in *Proc. Adv. Cryptol. (CRYPTO)*, in Lecture Notes in Computer Science, vol. 435. New York, NY, USA: Springer-Verlag, 1989, pp. 428–446.

[12] I. B. Damgård, "A design principle for hash functions," in *Proc. Adv. Cryptol. (CRYPTO)*, in Lecture Notes in Computer Science, vol. 435. New York, NY, USA: Springer-Verlag, 1989, pp. 416–427.

[13] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway, "A concrete security treatment of symmetric encryption," in *Proc. 38th Annu. Symp. Found. Comput. Sci.*, 1997, pp. 394–403.

[14] M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini, "Automated verification of block cipher modes of operation, an improved method," in *Foundations and Practice of Security*. Berlin, Germany: Springer, 2011, pp. 23–31.

[15] M. Gagné, P. Lafourcade, Y. Lakhnech, and R. Safavi-Naini, "Automated proofs of block cipher modes of operation," *J. Automated Reasoning*, vol. 56, no. 1, pp. 49–94, Jan. 2016.

[16] P. Rogaway. (2012). *Evaluation of Some Blockcipher Modes of Operation*. Accessed: Jan. 2024. [Online]. Available: https://www.cryptrec.go.jp/exreport/cryptrec-ex-2012-2010r1.pdf

[17] T. Yoshimura, K. Arai, H. Okazaki, and Y. Futa, "Formalization of security requirements and attack models for cryptographic hash functions in ProVerif," in *Proc. Int. Conf. Secur. Manag. (SAM)*, 2019, pp. 23–29.

[18] T. Mieno, T. Yoshimura, H. Okazaki, Y. Futa, and K. Arai, "Formal verification of Merkle–Damgård construction in ProVerif," in *Proc. Int. Symp. Inf. Theory Appl.*, 2020, pp. 602–606, doi: 10.34385/proc.65.E03-2.

[19] S. Marc, "New collision attacks on SHA-1 based on optimal joint local-collision analysis," in *Proc. Adv. Cryptol. (EUROCRYPT)*. Berlin, Germany: Springer, 2013, pp. 245–261.

[20] W. Xiaoyun and Y. Hongbo, "How to break MD5 and other hash functions," in *Proc. Adv. Cryptol. (EUROCRYPT)*. Berlin, Germany: Springer, 2005, pp. 19–35.

[21] M. Stevens, "Fast collision attack on MD5," Cryptol. ePrint Arch., Int. Assoc. Cryptologic Res., Paper 2006/104, 2006. Accessed: Nov. 2023. [Online]. Available: https://eprint.iacr.org/2006/104 and https://en.wikipedia.org/wiki/International_Association_for_Cryptologic_Research

[22] D. C. Christophe and C. Christian, "Finding SHA-1 characteristics: General results and applications," in *Proc. Adv. Cryptol. (ASIACRYPT)*. Berlin, Germany: Springer, 2006, pp. 1–20.

[23] Z. A. Al-Odat, S. U. Khan, and E. Al-Qtiemat, "A modified secure hash design to circumvent collision and length extension attacks," *J. Inf. Secur. Appl.*, vol. 71, Dec. 2022, Art. no. 103376, doi: 10.1016/j.jisa.2022.103376.

[24] B. Blanchet and M. Paiola, "Automatic verification of protocols with lists of unbounded length," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Berlin, Germany, 2013, pp. 573–584.

[25] A. Dax, R. Künnemann, S. Tangermann, and M. Backes, "How to wrap it up—A formally verified proposal for the use of authenticated wrapping in PKCS#11," in *Proc. IEEE 32nd Comput. Secur. Found. Symp. (CSF)*, Jun. 2019, pp. 6215–6262.

[26] M. Backes, G. Barthe, M. Berg, B. Gregoire, C. Kunz, M. Skoruppa, and S. Z. Beguelin, "Verified Security of Merkle–Damgård," in *Proc. 25th IEEE Comput. Secur. Found. Symp. (CSF)*, Cambridge, MA, USA, Jun. 2012, pp. 354–368.

[27] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *Proc. CRYPTO*, 2011, pp. 71–90.

[28] INRIA. *ProVerif Users Research Papers*. Accessed: Jan. 2024. [Online]. Available: https://bblanche.gitlabpages.inria.fr/proverif/proverif-users.html

[29] B. Blanchet, V. Cheval, and V. Cortier, "ProVerif with lemmas, induction, fast subsumption, and much more," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2022, pp. 69–86, doi: 10.1109/SP46214.2022.9833653.

[30] J. Wu, R. Wu, D. Xu, D. J. Tian, and A. Bianchi, "Formal model-driven discovery of Bluetooth protocol design vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2022, pp. 2285–2303, doi: 10.1109/SP46214.2022.9833777.
[31] A. Debant, S. Delaune, and C. Wiedling, "So near and yet so far—Symbolic verification of distance-bounding protocols," *ACM Trans. Privacy Secur.*, vol. 25, no. 2, pp. 1–39, May 2022.
[32] K. Bhargavan, V. Cheval, and C. Wood, "A symbolic analysis of privacy for TLS 1.3 with encrypted client hello," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Los Angeles, CA, USA, Nov. 2022, pp. 365–379, doi: 10.1145/3548606.3559360.
[33] M. Bougon, H. Chabanne, V. Cortier, A. Debant, E. Dottax, J. Dreier, P. Gaudry, and M. Turuani, "Themis: An on-site voting system with systematic cast-as-intended verification and partial accountability," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2022, pp. 397–410, doi: 10.1145/3548606.3560563.
[34] S. Hussain, M. Farooq, B. A. Alzahrani, A. Albeshri, K. Alsubhi, and S. A. Chaudhry, "An efficient and reliable user access protocol for Internet of Drones," *IEEE Access*, vol. 11, pp. 59688–59700, 2023, doi: 10.1109/ACCESS.2023.3284832.
[35] J. Guan, H. Li, H. Ye, and Z. Zhao, "A Formal analysis of the FIDO2 protocols," in *Computer Security—ESORICS*. Cham, Switzerland: Springer, 2022, pp. 3–21, doi: 10.1007/978-3-031-17143-7.
[36] C. Xu, W. Wei, and S. Zheng, "Efficient mobile RFID authentication protocol for smart logistics targets tracking," *IEEE Access*, vol. 11, pp. 4322–4336, 2023, doi: 10.1109/ACCESS.2023.3234959.
[37] C. Jacomme, E. Klein, S. Kremer, and M. Racouchot, "A comprehensive, formal and automated analysis of the EDHOC protocol," in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, 2023, pp. 5881–5898.
[38] V. Cheval and I. Rakotonirina, "Indistinguishability beyond diff-equivalence in ProVerif," in *Proc. IEEE 36th Comput. Secur. Found. Symp. (CSF)*, Dubrovnik, Croatia, Jul. 2023, pp. 184–199.
[39] V. Cheval, C. Jacomme, S. Kremer, and K. Robert ünnemann, "SAPIC+: Protocol verifiers of the world, unite!" in *Proc. USENIX Secur. Symp. (USENIX Security)*, 2022, pp. 3935–3952.
[40] C. Boyd and M. Anish, *Protocols for Authentication and Key Establishment*, 2nd ed. Berlin, Germany: Springer, 2019.
[41] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, May 2021, pp. 777–795, doi: 10.1109/SP40001.2021.00008.
[42] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, "Handbook of applied cryptography," in *Hash Functions and Data Integrity*, 1st ed. CRC Press,Dec. 1996, ch. 9, pp. 334–335. Accessed: Jan. 2024. [Online]. Available: https://cacr.uwaterloo.ca/hac/about/chap9.pdf
[43] X. Lai and J. L. Massey, "Hash functions based on block ciphers," in *Proc. EUROCRYPT*, in Lecture Notes in Computer Science, vol. 658. Cham, Switzerland: Springer, 1992, pp. 53–66.
[44] S. Vaudenay, "Security flaws induced by CBC padding—Applications to SSL, IPSEC, WTLS…" in *Proc. EUROCRYPT*, vol. 2332. Cham, Switzerland: Springer, 2002, pp. 534–546.
[45] B. Möller, T. Duong, and K. Kotowicz. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. Accessed: Jan. 2024. [Online]. Available: https://www.openssl.org/ bodo/ssl-poodle.pdf
[46] B. Blanchet, "Using horn clauses for analyzing security protocols," in *Formal Models and Techniques for Analyzing Security Protocols* (Cryptology and Information Security Series), vol. 5. IOS Press, 2011, pp. 86–111. [Online]. Available: https://bblanche.gitlabpages.inria.fr/proverif/publications/BlanchetBook09.html
[47] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. 14th IEEE Comput. Secur. Found. Workshop*, Jun. 2001, pp. 82–96.
[48] B. Blanchet, "From secrecy to authenticity in security protocols," in *Proc. 9th Int. Static Anal. Symp. (SAS)*, in Lecture Note on Computer Science, vol. 2477, 2002, pp. 342–359.
[49] B. Blanchet, "Automatic proof of strong secrecy for security protocols," in *Proc. IEEE Symp. Secur. Privacy, Proceedings.*, May 2004, pp. 86–100.
[50] B. Blanchet, M. Abadi, and C. Fournet, "Automated verification of selected equivalences for security protocols," in *Proc. 20th Annu. IEEE Symp. Log. Comput. Sci. (LICS)*, Jun. 2005, pp. 331–340.
[51] M. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," NIST, Gaithersburg, MS, USA, NIST Special Publication 800-38A, 2001 Edition, 2001. Accessed: Jan. 2024. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-38a/final
[52] NICT. *Cryptographic Protocol Verification Portal*. Accessed: Jan. 2024. [Online]. Available: https://crypto-protocol.nict.go.jp/index_en.html
[53] S. Lucks, "The saturation attack—A bait for Twofish," in *Proc. 8th Int. Workshop Fast Softw. Encryption (FSE)*, in Lecture Note on Computer Science, vol. 2355, 2001, pp. 1–15.
[54] X. Lai, "Higher order derivatives and differential cryptanalysis," in *Communications and Cryptography*. USA: Springer, 1994, pp. 227–233. [Online]. Available: https://scholar.google.com/citations?view_op=view_citation&hl=en&user=B67-NyQAAAAJ&citation_for_view=B67-NyQAAAAJ:qjMakFHDy7sC
[55] M. Luby, *Pseudorandomness and Cryptographic Applications*. Princeton, NJ, USA: Princeton Univ. Press, 1996, pp. 112–114.
[56] *Secure Hash Standard*, document FIPS 180-1, Nat. Inst. Sci. Technol., Federal Inf. Process. Standard (FIPS), Gaithersburg, MS, USA, 1995. Accessed: Jan. 2024. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-1
[57] R. L. Rivest, *The MD5 Message Digest Algorithm*, document RFC 1321, Apr. 1992.

TAKEHIKO MIENO (Member, IEEE) graduated from Shinshu University, Japan, and the M.E. degree from Shinshu University in 2023, where he is currently pursuing the degree with the Graduate School of Medicine, Science and Technology. He is with EPSON AVASYS Corporation. He was worked on the security development of many printers, sensors firmware, and drivers. His research interests include information security and formal verification.

HIROYUKI OKAZAKI received the B.E., M.E., and D.E. degrees in communication engineering from Kyoto Institute of Technology, in 1999, 2001, and 2004, respectively. He is currently an Associate Professor with the Graduate School, Division of Science and Technology, Shinshu University. His research interests include cryptology, information security, and formal verification of cryptography.

KENICHI ARAI received the B.E., M.E., and Dr.E. degrees from Shinshu University, Japan, in 2004, 2006, and 2010, respectively. He is currently an Associate Professor with the School of Information and Data Sciences, Nagasaki University. His research interests include information security and formal verification.

YUICHI FUTA received the D.E. degree in engineering from Shinshu University, in 2012. He was with Panasonic Corporation, from 1998 to 2013, and was engaged in research and development of information security and digital right management systems. He is currently a Professor with the School of Computer Science, Tokyo University of Technology. His research interests include information security and formal verification of security proofs.