BCiC: A System for Code Authentication and Verification

Nathan Whitehead and Martín Abadi

Department of Computer Science
University of California, Santa Cruz
{nwhitehe,abadi}@cs.ucsc.edu

Abstract. We present BCiC, a system for verifying and authenticating code that combines language-based proof methods with public-key digital signatures. BCiC aims to augment the rigor of formal proofs about intrinsic properties of code by relying on authentication and trust relations. BCiC integrates the Binder security language with the Calculus of (Co)Inductive Constructions (CiC). In this respect, it is a descendant of our previous logic BLF, which was based on LF rather than CiC. This paper focuses on the architecture and implementation of BCiC. In addition to a logical inference engine, the design most notably includes a network communication module for the efficient exchange of logical facts between hosts, and a cryptography module for generating and checking signatures. The implementation cooperates with the Open Verifier, a state-of-the-art system for proof-carrying code with modular checkers.

1 Introduction

Modern software comes from a multitude of sources, and it often comes in pieces. Some applications dynamically link to libraries, some are extended with applets or plug-in modules, and others can be automatically updated. In every case, policies and mechanisms for establishing trust in new code are essential. When the new code is signed with a public-key digital signature, trust in the code may be based on trust in its signer. More generally, trust in the code may result from authenticating the source of the code. However, such trust has limits: many signers are unknown or only partly known to the consumers of code, and even reputable signers make mistakes. Therefore, evaluating the code itself and its properties is also important. It can yield fundamental safety guarantees, as in Java bytecode verification [17], and it need not burden code consumers with proofs of code properties, as research on proof-carrying code (PCC) [18] demonstrates. (With PCC, code comes accompanied by safety proofs, and consumers need only check, not generate, the proofs.) Nevertheless, formal specification and analysis of code remain difficult and often incomplete, particularly when we go beyond basic memory-safety guarantees.

In this paper we present an approach and a system for establishing trust in code that combine signatures and proofs. We define a policy language that allows references to signed assertions and supports reasoning about trust in the signers. The policy language can also express theorems about code properties, and supports reasoning about the correctness of proofs of the theorems. The final decision to run code, and what privileges to give to the code, may require both signatures from trusted parties and direct proofs of code safety. For instance, it may require a partial proof of safety with trusted, signed assertions as hypotheses.

Specifically, we introduce BCiC, a system for verifying and authenticating code that combines language-based proof methods with public-key digital signatures. BCiC aims to augment the rigor of formal proofs about intrinsic properties of code by relying on authentication and trust relations. BCiC integrates the Binder security language [10] with the Calculus of (Co)Inductive Constructions (CiC) [8]. In this respect, it is a descendant of our previous logic BLF [22], which was based on LF [13] rather than CiC. Here we go beyond our work on BLF by designing and building a concrete system. In addition to a logical inference engine, the design most notably includes a network communication module for the efficient exchange of logical facts between hosts, and a cryptography module for generating and checking signatures. The implementation cooperates with the Open Verifier [6], a state-of-the-art system for proof-carrying code with modular checkers.

After considering previous and related work in Section 2, we give a short example in Section 3 and present a high-level overview of our system in Section 4. In Section 5 we define the syntax and logical meaning of policies, and describe the implementation of the logical inference engine. In Section 6 we present two important components of the system in more detail, the cryptography module and the network module. In Section 7 we describe the integration of BCiC with the Open Verifier. We conclude with some comments on future work in Section 8.

2 Related Work and Background

Many existing systems combine reason and authority in some way. Checking the validity of an X.509 certificate involves a combination of trusting principals and reasoning about the transitivity of certification. Environments that execute network code often combine static typechecks of the code with signature checks [11,15,17]. These systems can verify only fixed, simple properties of the code. PCC allows more interesting properties to be checked, but existing work on PCC [1,3,18,19] assumes that properties and proof rules are fixed ahead of time between the code producer and the code consumer; they also do not support signatures in their reasoning. Our previous paper [22] contains further discussion of related work, in particular of research on proof-carrying authentication [2, 4, 16]. For the sake of brevity we do not reproduce that material here; it is somewhat less relevant for the present paper.

BLF is a logic for authorizing code that combines reason and authority [22]. The logic in our new system is similar to BLF but, instead of combining Binder [10] and LF [13], it combines Binder and CiC, the Calculus of (Co)Inductive Constructions [8], used in the Coq tool [5, 21]. We switched to CiC in order to allow the use of Coq for theorem proving. We have found that inductive definitions for data structures yield significant advantages in proofs. The Coq environment also allows a high degree of organization and automation, and is thus friendly to large-scale theorem proving efforts. Our formal description in Section 5 is an updated version of our previous presentation, adapted to the new choice of logical framework.

Although the change in logical framework is significant, the primary difference between our current and previous work is that BCiC has a definite system architecture and a concrete realization whereas BLF does not. Previously we implemented BLF in an abstract way. All computation was contained in one machine and the interactions between hosts were simulated. Public-key digital signatures were considered abstract terms; no actual cryptographic algorithms were used. Similarly, logical formulas were manipulated abstractly. In our present work the implementation is concrete. Signatures and logical formulas have concrete representations as binary bit strings in a standardized format. Hosts communicate with one another over the Internet in order to share data, following the formal semantics of the import and export functions in Section 5. We have also connected our system with an actual PCC framework, the Open Verifier, as described in Section 7.

The implementation of our communication structure, in which pairs of hosts synchronize and exchange new information, is inspired by work on replicated databases and database synchronization [9, 14]. In our case, the database which is being synchronized is a set of logical statements.

3 An Example

This section motivates and introduces some components of the system through an example.

Suppose that Alice is a user who requires that every program she executes neither access memory incorrectly nor use too many resources. There may be a relatively straightforward way to prove memory safety for the programs of interest, but not one for characterizing resource usage. Moreover, excessive resource usage may not be viewed as very harmful when there is someone to blame and perhaps bill. Accordingly, Alice may want proofs for memory safety but may trust Bob on resource usage. Alice constructs a policy that includes:

```
use R in
  forallobj P:program
  mayrun(P) :- sat(safe P), economical(P)
  economical(P) :- Bob says economical(P)
end
```

The first line indicates that `R` applies, as an environment. This environment is a set of constructors and proof rules that define the syntax of programs and the rules employed for proving memory safety. It is specific to a particular programming language and proof methodology. For instance, `R` may contain the following snippet, which defines standard constructs for memory access:

```
mem : Type
sel : mem -> val -> val
upd : mem -> val -> val -> mem
```

The second line of the policy (`forallobj P:program`) is a universal quantification over all programs `P`. The first clause indicates that Alice believes that she may execute a program P if there is a proof that P is memory safe and she thinks that P is economical. The second clause reflects Alice's trust in Bob. In more complex examples, other clauses may provide other ways of establishing economical(P). The operator says, from Binder, represents assertions by principals. The sat construct is a special logic predicate that holds when there is a CiC proof of its argument. The other predicates (mayrun and economical) are user-defined predicates.

In turn, Bob trusts Charlie to write only economical programs, and has in his policy:

```
use R in
  forallobj P:program
  economical(P) :- Charlie says mine(P)
end
```

where mine is another user-defined predicate.

Suppose further that Charlie supplies a program P0 that Alice wishes to execute. Charlie produces a CiC proof, Pf, of memory safety of the program. Charlie publishes his proof by asserting proof(Pf), specifically by typing the command bcicclient assert proof(Pf). The predicate proof does not have any built-in logical meaning; it simply serves for introducing the proof Pf. Similarly, Charlie asserts mine(P0).

Alice, Bob, and Charlie all run BCiC servers in the background. When the servers are set up, they are given the address of an existing server. From that point, they synchronize and receive a list of all other known servers. Once connected, they occasionally choose other servers with which to synchronize. After sufficient synchronization, Alice can deduce economical(P0) and Charlie says proof(Pf). After the logic inference engine checks the proof Pf, Alice obtains sat(safe P0). Now when Alice queries mayrun(P0), she receives the answer "yes" and is prepared to run P0.

4 Overview

Although our system must implement the logic presented in the next section in order to support reasoning about signatures and proofs (like the reasoning in the example), the system is much more than a bare implementation of the logic. In such a bare implementation, for instance, signed statements may simply be logical expressions—appropriate for initial experimentation but not much else. In order to be useful, signed statements must have concrete, secure digital representations. Thus, in our system, signatures employ cryptography; they are unforgeable and tamper-evident. Furthermore, our system deals with communication over an insecure network. The network module should minimize the need for manual user intervention for synchronization.

The implementation has several parts:

– The *parser* understands the syntax of the logic and can translate between textual and abstract syntax tree representations.
– The *logic interpreter* performs deductions in the logic from a set of given statements to produce a larger (but still finite) set of deduced statements.
– The *cryptography module* implements the necessary cryptographic operations, such as generating and checking signatures.

– The *network module* can communicate statements over the network.
– The *user interface* accepts textual input from the user and determines which action should be taken. The user interface is a simple command-line utility that communicates with an existing daemon over a secure local socket.
– The *supervisor* is in charge of coordinating the global behavior of the program. It loads existing databases of statements, decides when to communicate on the network, sign statements, draw inferences, and accept input from the user.
– The *policy* gives rules for deciding when code should be executed, who to trust initially about what, and so forth.
– The *database* holds all known true facts from any source.

Figure 1 shows the organization of these components in the system. Boxes represent code modules, circles represent data. The figure also shows the Open Verifier; this part is explained in Section 7.

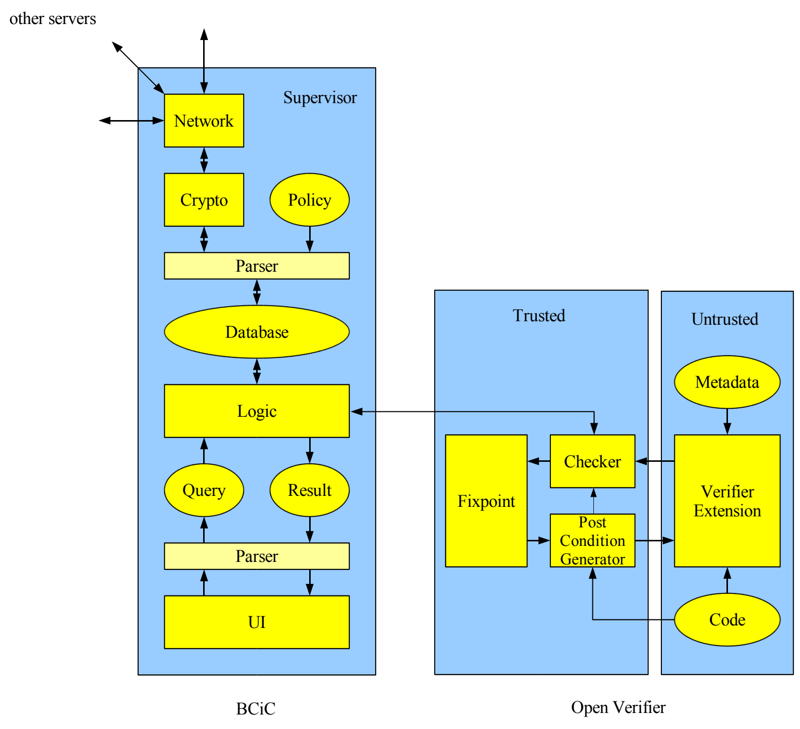

Fig. 1. System components

5 BCiC's Policy Language

This section presents the syntax and semantics of our policy language in a somewhat abbreviated fashion. Details can be found in the appendix. The interested reader is also encouraged to refer to corresponding descriptions for BLF [22] for additional explanations. What we describe here, however, should suffice for understanding the rest of this paper.

5.1 Formal Presentation of the Language

The policy language has both a user syntax (defined in Figure 2 of the appendix) and an internal syntax (in Figure 4). The user syntax allows proof rules to be stated in environments of the form `use R in ... end`. On the other hand, the internal syntax records proof rules at every point where they are employed, as extra parameters rather than with `use`. A simple annotation function translates from the user syntax into the internal syntax (Figure 3). Rulesets (type signatures) are typed according to the identifiers that they define. In this respect, BCiC is more constrained than BLF. Therefore, when one quantifies over rulesets, only rulesets with the proper definitions are considered.

The logical meaning of policies is given by proof rules (in Figure 5). These proof rules rely on the CiC typing relation (written $\vdash_{CiC}$). They also rely on conversions to normal form, as calculated by Coq. In other respects, the proof rules are a fairly ordinary logical system for standard logical constructs. We formulate them as sequent deductions [12].

The import function is used for determining the logical meaning of signed statements received over the network. It is a partial function that takes a key and a formula and returns a new formula. Our method for importing statements follows Binder. An atomic formula A signed with a key U is imported as U `says` A. It is also possible to export and import some (but not all) non-atomic formulas. A clause can be imported from U only if the head of the clause is not already quoted with `says`. If the original clause is H :- B, then the imported clause will be U `says` H :- B' where B' is the original body with every formula F without a `says` prefix replaced with U `says` F. In terms of g-formulas and d-formulas as used in the formal syntax, a g-formula G without a `says` prefix is translated to U `says` G. A g-formula of the form V `says` G remains unchanged in translation. A d-formula D without `says` gets translated to U `says` D, while d-formulas of the form V `says` D are untranslatable.

5.2 The Logic Interpreter

The logic module is responsible for managing the fact database and responding to queries. Initially the fact database contains only the facts in the local policy. After synchronizing with other hosts and exchanging signed statements, the fact database will grow. The main job of the logic interpreter is finding all possible logical deductions within the database and adding them to the database.

This method of answering queries is bottom-up evaluation. The bottom-up approach has the advantage that it is simple and clearly exhaustive. In contrast, the termination of top-down inference for Datalog (and therefore for Binder) requires tabling, which can give rise to subtle issues [7, 20]. Moreover, bottom-up evaluation immediately offers a convenient memoizing capability. Although bottom-up evaluation can require more time and space than top-down evaluation, we believe that bottom-up evaluation is practical for our application.

The basic operation of the logic interpreter is as for BLF except for term normalization. The interpreter repeatedly examines the database and systematically attempts to apply every term to every other term. If the application succeeds, then the new result is normalized and added to the database and the process repeats. Normalization is done in a module that applies the rules for CiC using code borrowed from Coq.

Although the logic interpreter always terminates on pure Datalog policies, it need not terminate on policies that make a non-trivial use of CiC. Infinite loops may happen when applying one dependent type to another results in a more complicated type. Fortunately, we have not seen this behavior in practice. Moreover, it should be possible to define a syntactic restriction that guarantees termination. We have such a restriction in BLF, and believe that we know how to port it to BCiC if it proves necessary.

6 Other System Modules

This section describes the cryptography module and the network module in more detail.

6.1 The Cryptography Module

The cryptography module is based on Xavier Leroy's library for OCaml, `cryptokit`, a library that provides cryptographic primitives. We use these primitives for generating keys, for signing, and for verifying signatures. We rely on RSA signatures with a key length of 1024 bits, and we apply SHA-1 hashing before signing. Each signed logical statement is accompanied by public-key information about the signer. Verifying that Alice signs statement `X` leads to an entry in the fact database, with the formula `Alice says X`. We serialize and deserialize statements using the `Marshal` standard library functions of OCaml.

When keys are not managed securely, the integrity of every signature is suspect. Therefore, following standard practices, we store secret keys in encrypted form, keyed to the hash of a passphrase supplied by the user. We use AES encryption and SHA-1 hashes for storing secret RSA keys.

6.2 The Network Module

The network module is only in charge of communicating signed statements between hosts, not determining their logical meaning. When new statements become available, the logic inference module must decide how they are to be interpreted. When the logic inference algorithm adds new unquoted conclusions (that is, formulas without `says`) to the database, the cryptography module creates new signatures and stores them in the database, and the network module communicates them to other hosts.

Network communication is done using TCP/IP connections on a specific port. Users may leave a Unix daemon running at all times waiting for connections, if they wish.

When two BCiC nodes connect, they follow a protocol to decide which statements are known to one and not the other. These statements are then transmitted over the connection. Connections can be scheduled to occur automatically with randomly chosen partners at regular intervals, or can be requested manually by users. Full-time servers may run the daemon and automatically communicate with one another, while individual client machines may rather connect to the nearest server sporadically at a user's request.

The most interesting aspect of the network module is the algorithm for coordinating updates. When nodes connect, they must decide who has new statements that need to be transmitted. Simply transmitting all the statements at every connection would be tremendously inefficient. A slightly more realistic possibility is a naive protocol in which each node hashes every statement in its database and communicates the entire set of hashes to the other node. Then it is easy for each node to decide which statements it must send. If n is the total number of statements known by both sides, then the naive protocol takes $O(n)$ steps per synchronization.

The naive protocol is likely not to be efficient enough in the long run. The size of the fact databases will steadily increase over time, if nothing else because expiration and revocation are rare (and they are not even modeled explicitly in the logic, although the implementation deals with them). More specifically, we may estimate the performance of the naive protocol as follows. A large library may be composed of several hundred functions. The library provider may wish to declare some functions correct by assertion and to verify other, simpler functions. One way to do this is for the library provider to sign assertions for each of the functions separately. As new versions of library functions become available, new statements will be generated. A fairly typical Linux operating system in our lab currently uses 652 libraries and 2777 applications. If every library requires 100 statements and releases 10 major versions a year, with each version containing 10 function updates, and every application releases 10 new versions a year, then the database will initially contain 67977 statements and will increase by 92970 statements each year. After two years the naive protocol will be exchanging 5 Mb at each connection. Even if one reduces the number of statements at the expense of statement size by signing one large conjunction that contains statements for all the functions in a library release, the protocol will still be exchanging 2.7 Mb at each connection after two years.

There are many possible solutions to the synchronization problem. It is not too hard to imagine methods that record timestamps or remember which facts have already been communicated to other servers. We chose to implement a recursive divide-and-conquer protocol that does not require any extra storage outside the databases themselves. It is asymptotically efficient for small updates between large databases.

Our approach requires that every statement in every database be hashed and stored sorted by hash value. Our protocol synchronizes ranges of hash values between two databases. To synchronize two entire databases, the protocol is performed over the entire range of possible hash values. To synchronize all hash values between L and H, first both participants extract all hash values in their databases between L and H. Each list is encoded and hashed, then exchanged between the participants. A special token is used to represent the hash of the empty list. If the hash values agree, then both databases are already synchronized and the protocol terminates. If one hash is nonempty and one is empty, then the protocol terminates and the participant with the nonempty list knows they must transfer the contents of the list. If both hashes are nonempty and differ, then the range of hash values is split into two equal subranges, L to M and M to H. The protocol is then applied recursively on these two subranges. (We could also use more than two subranges, of course.) If n is the total number of statements known by both sides, and m is the maximum number of statements known by one side that are not known by the other, then an update takes $O(m \log n)$ steps.

Transmitting one large batch of data is often much faster than performing several rounds of communication to determine which parts of the data should be sent. By communicating the number of statements that were hashed at each exchange, the implementation can switch to the naive protocol when the number falls beneath a threshold. In experiments we found the optimal threshold to be approximately 1200 children for connections between computers in our lab and a laptop off campus. The network module of BCiC uses the naive protocol on databases of size 1200 or smaller, and uses the recursive protocol on larger databases.

7 Integrating BCiC with the Open Verifier

The Open Verifier [6] supports verifying untrusted code using modular verifiers. Programs are expressed in a low-level assembly language with formally specified semantics. Code producers may provide verification modules for creating proofs of code safety on demand, rather than actual proofs. Figure 1 illustrates the workings of the Open Verifier. The code, verifier extension module, and metadata on the right are untrusted and provided by the code producer. The trusted components on the left (the fixpoint module, post-condition generator, and checker) communicate with the untrusted verifier extension in order to generate a conjunction of invariants with proofs.

In this section we explain how BCiC can be connected to the Open Verifier. We focus on what we have implemented: a scheme in which the Open Verifier can call BCiC. We also discuss, more briefly, a scheme in which BCiC can call the Open Verifier.

7.1 The Open Verifier calling BCiC

Supplementing the Open Verifier with BCiC makes verification with plug-in verifiers even more flexible. Instead of requiring that verifiers be able to prove code safety absolutely, we allow the verifiers to use assumptions that are trusted because they have been asserted by trusted authorities. This arrangement might be necessary for difficult safety properties. It also allows a verifier to prove something different than required if there is a trusted assumption that says that the property proved implies the required property. In particular, the verifier may do a “proof by typechecking”: it may typecheck a program, and a trusted assumption may declare that typechecked programs are safe.

In the normal operation of the Open Verifier, the fixpoint module collects invariants that must be verified. First the fixpoint module supplies an initial invariant to the post-condition generator. The strongest post-condition is calculated and then passed to the untrusted verifier extension, which responds with weaker invariants and proofs that they are indeed weaker. These proofs are checked using Coq by the checker module. The weaker invariants are collected in the fixpoint module, which continues to calculate a fixpoint of invariants by possibly sending more invariants back to the post-condition generator.

The connection between BCiC and the Open Verifier affects the communication between the post-condition generator, the untrusted verifier extension, and the checker. We have integrated BCiC so that when the Open Verifier decides that it needs justification for a weaker invariant, instead of the Open Verifier asking the extension directly, BCiC first checks its database of facts. If the statement already appears in the database, then the extension is never queried and the Open Verifier continues as if the justification were received. If the statement is not in the database, then the extension is asked for the justification as usual.

This scheme allows the BCiC database to short-circuit the interactive proof protocol at any point. Untrusted code can be asserted to be safe without any proof. In this case there must be an entry in the BCiC database that corresponds to the first query that the Open Verifier provides to the extension. In particular, this scheme handles "proofs by typechecking". When the extension can verify that the code typechecks but cannot justify the soundness of the typechecking rules, the soundness lemmas can appear in the BCiC database.

7.2 BCiC calling the Open Verifier

Currently the Open Verifier is limited to verifying a single, generic memory-safety property. This focus is reasonable in light of current verification techniques, but allowing signatures opens the door to handling other properties. BCiC can support reasoning about those properties, calling the Open Verifier when appropriate.

For this purpose, we envision a mechanism whereby the conclusions of the Open Verifier can be used as new facts in the BCiC database. More specifically, the conclusions of the Open Verifier are represented as logical facts in BCiC, with a new predicate `verified`. We are currently refining our design and implementation of this scheme, and a mechanism for running programs subject to BCiC policies.

8 Conclusion

In this paper we describe BCiC, a system for reasoning about code that can combine proofs of code properties and assertions from trusted authorities. We present the underlying logic, show the architecture of the system itself, and describe our method of integration with the Open Verifier. Going from an abstract logic to an actual system requires a fair amount of work and a number of significant design choices. Although our system is still experimental, we believe that it shows one possible avenue for progress in code authentication and verification.

So far we have used BCiC for experimenting with small programs created to exercise various features of theorem provers. Perhaps the most important remaining work is to apply BCiC to large, useful programs. Clearly BCiC can handle those programs—at least in uninteresting ways—since it subsumes technologies that scale well (typechecking, public-key digital signatures). Going further, it would also be interesting to deploy the system in an environment where many users may place different amounts of trust in many programs. This deployment would allow more experimentation with policies and would test the effectiveness of the network protocol.

Acknowledgments

Thanks to George Necula for helpful discussions and allowing us access to the source code of the Open Verifier. Thanks to Katia Hayati and Bogdan Warinschi for providing feedback on early drafts of this paper. This work was supported in part by the National Science Foundation under Grants CCR-0204162 and CCR-0208800.

References

1. Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 247–258, June 2001.
2. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 52–62, November 1999.
3. Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 243–253, January 2000.
4. Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the Web. In *Proceedings of the 11th USENIX Security Symposium 2002*, pages 93–108, 2002.
5. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
6. Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. The Open Verifier framework for foundational verifiers. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, pages 1–12, 2005.
7. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
8. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.
9. A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Symposium on Principles of Distributed Computing*, pages 1–12, August 1987.
10. John DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 105–113, May 2002.
11. ECMA. Standard ECMA-335: Common Language Infrastructure, December 2001. Available on-line at: http://msdn.microsoft.com/net/ecma/.
12. Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1990. http://nick.dcs.qmul.ac.uk/ pt/stable/Proofs+Types.html.
13. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
14. JoAnne Holliday, Robert C. Steinke, Divyakant Agrawal, and Amr El Abbadi. Epidemic algorithms for replicated databases. *IEEE Transactions on Knowledge Data Engineering*, 15(5):1218–1238, 2003.

15. Sebastian Lange, Brian LaMacchia, Matthew Lyons, Rudi Martin, Brian Pratt, and Greg Singleton. *.NET Framework Security*. Addison Wesley, 2002.
16. Eunyoung Lee and Andrew W. Appel. Policy-enforced linking of untrusted components. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 371–374, September 2003.
17. T. Lindholm and F. Yellin. *The Java*$^{\mathrm{TM}}$ *Virtual Machine Specification*. Addison Wesley, 1997.
18. George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, 1997.
19. George C. Necula and Robert R. Schneck. A sound framework for untrusted verification-condition generators. In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science*, pages 248–260, July 2003.
20. P. Rao, K. Sagonas, T. Swift, D. S. Warren, and J. Freire. XSB: A system for efficiently computing well-founded semantics. In *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning*, volume 1265 of *Lecture Notes in Artificial Intelligence*, pages 430–440, Berlin, July 1997. Springer.
21. The Coq Development Team. The Coq proof assistant. http://coq.inria.fr/.
22. Nathan Whitehead, Martín Abadi, and George Necula. By reason and authority: A system for authorization of proof-carrying code. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 236–250, June 2004.

Appendix

This appendix contains Figures 2 through 5. These figures provide details of the formal syntax and semantics of BCiC. Some additional background and informal explanations can be found with the formal presentation of BLF [22].

```
⟨var, rsvar, termvar, cicvar⟩ ::= ⟨identifier⟩

⟨policy⟩ ::= [ ⟨dform⟩. ]+

⟨predicate⟩ ::= ⟨identifier⟩

⟨principal⟩ ::= ⟨key⟩ | ⟨var⟩

⟨argument⟩ ::= ⟨identifier⟩ | ⟨key⟩ | ⟨var⟩ | ⟨expr⟩
    | ⟨ruleset⟩

⟨ruleset⟩ ::= ⟨rsvar⟩ | ⟨actualruleset⟩ | ⟨rsvar⟩; ⟨ruleset⟩

⟨actualruleset⟩ ::= ruleset( [ ⟨identifier⟩ : ⟨lfterm⟩. ]* )

⟨expr⟩ ::= ⟨termvar⟩ | ⟨cicvar⟩ | type | set | prop | ⟨expr⟩ ⟨expr⟩
    | ⟨expr⟩ → ⟨expr⟩ | {⟨cicvar⟩ : ⟨expr⟩} ⟨expr⟩
    | [⟨cicvar⟩ : ⟨expr⟩] ⟨expr⟩

⟨gform⟩ ::= ⟨atomic⟩ | ⟨gform⟩, ⟨gform⟩ | ⟨gform⟩; ⟨gform⟩
    | exists ⟨var⟩ ⟨gform⟩
    | existrules ⟨rsvar⟩ : ⟨rulesettype⟩ ⟨gform⟩
    | existsobj ⟨termvar⟩ : ⟨expr⟩ ⟨gform⟩
    | use ⟨ruleset⟩ in ⟨gform⟩ end

⟨dform⟩ ::= ⟨atomic⟩ | ⟨dform⟩, ⟨dform⟩
    | ⟨dform⟩ :- ⟨gform⟩ | forall ⟨var⟩ ⟨dform⟩
    | forallrules ⟨rsvar⟩ : ⟨rulesettype⟩ ⟨dform⟩
    | forallobj ⟨termvar⟩ : ⟨expr⟩ ⟨dform⟩
    | use ⟨ruleset⟩ in ⟨dform⟩ end

⟨atomic⟩ ::= [ ⟨principal⟩ says ] sat(⟨expr⟩)
    | [ ⟨principal⟩ says ] believe(⟨expr⟩)
    | [ ⟨principal⟩ says ] ⟨predicate⟩
          ( [ ⟨argument⟩ [ , ⟨argument⟩ ]* ] )

⟨rulesettype⟩ ::= [ ⟨identifier⟩ [ , ⟨identifier⟩ ]* ]
```

Fig. 2. Syntax

$$[A, B]_R = [A]_R, [B]_R$$
$$[A; B]_R = [A]_R; [B]_R$$
$$[D \text{ :- } G]_R = [D]_R \text{ :- } [G]_R$$
$$[\texttt{forall}\ x\ D]_R = \texttt{forall}\ x\ [D]_R$$
$$[\texttt{exists}\ x\ G]_R = \texttt{exists}\ x\ [G]_R$$
$$[\texttt{forallrules}\ r : T\ D]_R = \texttt{forallrules}\ r : T\ [D]_R$$
$$[\texttt{existrules}\ r : T\ G]_R = \texttt{existrules}\ r : T\ [G]_R$$
$$[\texttt{forallobj}\ x : T\ D]_R = \texttt{forallobj}'\ R\ x : T\ [D]_R$$
$$[\texttt{existsobj}\ x : T\ G]_R = \texttt{existsobj}'\ R\ x : T\ [G]_R$$
$$[\texttt{use}\ R'\ \texttt{in}\ A\ \texttt{end}]_R = [A]_{R;R'}$$
$$[P\ \texttt{says}\ X]_R = P\ \texttt{says}\ [X]_R$$
$$[\texttt{sat}(T)]_R = \texttt{sat}'(R, T)$$
$$[\texttt{believe}(T)]_R = \texttt{believe}'(R, T)$$
$$[\mathsf{P}(\alpha_1, \alpha_2, \ldots, \alpha_n)]_R = \mathsf{P}(\alpha_1, \alpha_2, \ldots, \alpha_n)$$

Fig. 3. Annotation function

$$
\begin{aligned}
\langle\text{gform}\rangle ::= {} & \langle\text{atomic}\rangle \ \mid\ \langle\text{gform}\rangle, \langle\text{gform}\rangle \ \mid\ \langle\text{gform}\rangle; \langle\text{gform}\rangle \\
& \mid\ \texttt{exists}\ \langle\text{var}\rangle\ \langle\text{gform}\rangle \\
& \mid\ \texttt{existrules}\ \langle\text{rsvar}\rangle : \langle\text{rulesettype}\rangle\ \langle\text{gform}\rangle \\
& \mid\ \texttt{existsobj}'\ \langle\text{ruleset}\rangle\ \langle\text{termvar}\rangle : \langle\text{expr}\rangle\ \langle\text{gform}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{dform}\rangle ::= {} & \langle\text{atomic}\rangle \ \mid\ \langle\text{dform}\rangle, \langle\text{dform}\rangle \\
& \mid\ \langle\text{dform}\rangle \text{ :- } \langle\text{gform}\rangle \ \mid\ \texttt{forall}\ \langle\text{var}\rangle\ \langle\text{dform}\rangle \\
& \mid\ \texttt{forallrules}\ \langle\text{rsvar}\rangle : \langle\text{rulesettype}\rangle\ \langle\text{dform}\rangle \\
& \mid\ \texttt{forallobj}'\ \langle\text{ruleset}\rangle\ \langle\text{termvar}\rangle : \langle\text{expr}\rangle\ \langle\text{dform}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle\text{atomic}\rangle ::= {} & [\ \langle\text{principal}\rangle\ \texttt{says}\ ]\ \texttt{sat}'(\langle\text{ruleset}\rangle, \langle\text{expr}\rangle) \\
& \mid\ [\ \langle\text{principal}\rangle\ \texttt{says}\ ]\ \texttt{believe}'(\langle\text{ruleset}\rangle, \langle\text{expr}\rangle) \\
& \mid\ [\ \langle\text{principal}\rangle\ \texttt{says}\ ]\ \langle\text{predicate}\rangle \\
& \qquad \texttt{(}\ [\ \langle\text{argument}\rangle\ [\,, \langle\text{argument}\rangle\,]^*\ \ ]\ \texttt{)}
\end{aligned}
$$

$$\vdots$$

Fig. 4. Internal syntax

$$\frac{}{A, \Gamma \Rightarrow \Delta, A} \quad A \text{ is atomic}$$

$$\frac{\phi(\Gamma) \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \phi(\Delta)}{\Gamma \Rightarrow \Delta} \quad \phi \text{ is a permutation}$$

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma, D \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, G} \qquad \frac{\Gamma, D, D \Rightarrow \Delta}{\Gamma, D \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G, G}{\Gamma \Rightarrow \Delta, G}$$

$$\frac{D_1, D_2, \Gamma \Rightarrow \Delta}{(D_1, D_2), \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G_1 \quad \Gamma \Rightarrow \Delta, G_2}{\Gamma \Rightarrow \Delta, (G_1, G_2)}$$

$$\frac{\Gamma \Rightarrow \Delta, G \quad D, \Gamma \Rightarrow \Delta}{D \text{ :- } G, \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, G_1, G_2}{\Gamma \Rightarrow \Delta, (G_1; G_2)}$$

$$\frac{D[A/x], \mathtt{forall}\ x\ D, \Gamma \Rightarrow \Delta}{\mathtt{forall}\ x\ D, \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, \mathtt{exists}\ x\ G, G[A/x]}{\Gamma \Rightarrow \Delta, \mathtt{exists}\ x\ G}$$

$$\frac{D[O/x], \mathtt{forallobj}'\ R\ x : T\ D, \Gamma \Rightarrow \Delta \quad R \vdash_{CiC} O : T}{\mathtt{forallobj}'\ R\ x : T\ D, \Gamma \Rightarrow \Delta}$$

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{existsobj}'\ R\ x : T\ G, G[O/x] \quad R \vdash_{CiC} O : T}{\Gamma \Rightarrow \Delta, \mathtt{existsobj}'\ R\ x : T\,.\,G}$$

$$\frac{D[R/r], \mathtt{forallrules}\ r\ D, \Gamma \Rightarrow \Delta}{\mathtt{forallrules}\ r\ D, \Gamma \Rightarrow \Delta} \qquad \frac{\Gamma \Rightarrow \Delta, \mathtt{existrules}\ r\ G, G[R/r]}{\Gamma \Rightarrow \Delta, \mathtt{existrules}\ r\ G}$$

$$\frac{R \vdash_{CiC} O : T}{\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, T)} \qquad \frac{\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, \{x : T\}B) \quad R \vdash_{CiC} O : T}{\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, B[O/x])}$$

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, T), \mathtt{sat}'(R, T)}{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, T)}$$

$$\frac{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, \{x : T\}B) \quad R \vdash_{CiC} O : T}{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, B[O/x])}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, \{x : T\}B) \quad \Gamma \Rightarrow \Delta, \mathtt{sat}'(R, T) \\ x \text{ does not occur in } B\end{array}}{\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, B)}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, \{x : T\}B) \quad \Gamma \Rightarrow \Delta, \mathtt{believe}'(R, T) \\ x \text{ does not occur in } B\end{array}}{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, B)}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, T') \\ T \text{ and } T' \text{ have the same normal form in ruleset } R\end{array}}{\Gamma \Rightarrow \Delta, \mathtt{sat}'(R, T)}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, T') \\ T \text{ and } T' \text{ have the same normal form in ruleset } R\end{array}}{\Gamma \Rightarrow \Delta, \mathtt{believe}'(R, T)}$$

Fig. 5. Proof rules