CLEI electronic journal, Volume 24, Number 3, Paper 0 , December 2021

Set-Based Models for Cryptocurrency Software

Gustavo Betarte
Universidad de la Repblica, Facultad de Ingeniera,
Montevideo, Uruguay
gustun@fing.edu.uy

and

Maximiliano Cristiá
CIFASIS and Universidad Nacional de Rosario
Rosario, Argentina
cristia@cifasis-conicet.gov.ar

and

Carlos Luna
Universidad de la Repblica, Facultad de Ingeniera,
Montevideo, Uruguay
cluna@fing.edu.uy

and

Adrián Silveira
Universidad de la Repblica, Facultad de Ingeniera,
Montevideo, Uruguay
adrians@fing.edu.uy

and

Dante Zanarini
CIFASIS and Universidad Nacional de Rosario
Rosario, Argentina
zanarini@cifasis-conicet.gov.ar

Abstract

Formal methods (FM) are mathematics-based software development methods aimed at producing "code for a nuclear power reactor". That is, due application of FM can produce bug-free, zero-defect, correct-by-construction, guaranteed, certified software. However, the software industry seldom use FM. One of the main reasons for such a situation is that there exists the perception (which might well be a fact) that FM increase software costs. On the other hand, FM can be partially applied thus producing high-quality software, although not necessarily bug-free.

In this paper we outline some FM related techniques whose application the cryptocurrency community should take into consideration because they could bridge the gap between "loose web code" and "code for a nuclear power reactor". We include relevant case studies in the area of cryptocurrency.

Keywords– Formal Methods, Set-Based Models, Verification, Cryptocurrency, Consensus Protocols, Ethereum VM.

1 Introduction

Given that cryptocurrency software deals with virtual money, software errors can produce irreparable loses. Furthermore, they are a valuable target of highly skilled attackers. Therefore, if a cryptocurrency software has an exploitable vulnerability chances are that attackers will eventually use it to steal money. In fact, some attacks have already been mounted against cryptocurrency software causing irreparable loses of money and credibility (e.g. [1]).

Note that cryptocurrency software face a more complex problem than traditional banking systems. If your homebanking account is hacked your money can be (potentially) wired transferred to one or more *bank accounts* anywhere on Earth. But these accounts have registered owners who can be duly prosecuted, although some times you will not get your money back. If the same happens in the cryptocurrency world you will not have a registered owner and so prosecution will be harder, if possible.

Hence, software errors in the cryptocurrency world are potentially more costly than the same class of errors in traditional banking systems. Therefore, the quality of cryptocurrency software should be, at least, one "level" higher than that of banking software. Although banking software is not (always) "loose web code", it certainly is not "code for a nuclear power reactor".

For these reasons the cryptocurrency community is seeking for approaches, methods, techniques and development practices that can reduce the chances of the presence of either errors or vulnerabilities[1]. The traditional banking system has less incentives to pursue high levels of software quality and thus to seek better development practices.

One such approach is the application of Formal Methods (FM) to software construction. FM are development methods based on mathematics and logic. They have been around in academia for at least 50 years. There have been undergraduate courses around the globe teaching FM for decades. A number of scientific journals and international conferences are devoted to the study, progess and application of FM. FM is the working field of hundreds of researchers across the planet. The FM community has produced breakthrough results on fundamental aspects of Computer Science and some of its most prominent actors have earned international prizes such as the Turing Award. Leading companies such as Microsoft and Amazon hire software engineers or researchers on FM to produce or apply FM.

When FM are fully applied they produce "code for a nuclear power reactor". That is, due application of FM can produce bug-free, zero-defect, correct-by-construction, guaranteed, certified software. There are a number of critical or mission critical software systems that were developed with FM.

However, the software industry seldom use FM. There are several reasons for which industry is reluctant to use FM among which we can cite: a pervasive underrating of software quality (usually in favor of innovation or as the consequence of free distribution[2]); either the perception or the fact that FM severely increase costs and extend schedules; at times the sheer fact that managers do not have a clue about the existence of FM; and the difficulties of finding the right technical staff.

In this paper we present the case for the application of FM to cryptocurrency software. In Sect. 2 we present some guidelines for the adoption of FM in cryptocurrency software projects. We argue that set-based formal modeling (or specification), simulation, prototyping and automated proof can be applied before considering more powerful approaches such as code formal verification. Hence, in Sect. 3 we show excerpts of a set-based formal specification of a consensus protocol and in Sect. 4 of the Ethereum Virtual Machine (EVM). In Sect. 5 we show that prototypes can be generated from these formal models and simulations can be run on them. Then, we show that test cases can be generated from the same models and how automated proof can be used to evaluate the correctness of these models.

2 Guidelines for Formal Methods Adoption

This section presents guidelines that the cryptocurrency community can consider for the adoption of FM. It starts with a brief and broad presentation of FM.

2.1 Formal methods in a nutshell

As we have said, FM is a class of software development techniques based on mathematics and formal logic that can be applied to different development activities and phases[3]. The main goal of FM is to improve the confidence on the correctness of a program. Program correctness is usually defined as follows: *a program is correct if it verifies its specification.* The word 'verifies' can be replaced by 'satisfies', 'refines', 'respects',

[1] Actually, the presence of certain errors lead to the existence of vulnerabilities.

[2] That is: Why should I produce bug-free software if I am not sure anyone will use it? Why are you asking me for a warranty if I am giving you this program for free?

[3] FM can be applied to hardware systems, too.

etc. In any case, the fundamental idea is that you have a program, P, and its specification, S, and a FM provides you a way to ensure, with different levels of confidence, that P behaves as S states.

In FM the specification or model, S plays a central role:

- S should be written from the requirements, not from the program; ideally S should be written before P is developed.
- S is a formal description; that is, in a sense, S is a mathematical model or formula.
- S represents a family of programs: from S you can get a number of different programs; S does not fix a particular implementation.
- S should be simpler and shorter than P but, fundamentally, it should be more evidently correct than P. That is, when you read S you should feel you are reading something that obviously describes what you want.
- S should abstract away as many implementation details as possible. Implementing a program entails giving enough details as to a computer can execute it. A huge number of these details are nonexistent in the requirements. For example, at the requirements level one simply says "if the customer is a new one, then add it to the data base", while a C implementation might use a singly-linked list to store customers and so checking if the customer exists and adding it requires pointer arithmetic. In this case, pointer arithmetic is an implementation detail that should not be visible in S.
- S is usually a functional specification but it can also be a security specification (or whatever other non-functional requirement). Then, FM can be used to verify that P correctly does what it is supposed to do and that it does that securely.

Then, the application of FM starts by writing a formal specification of your software. It does not mean you must write a complete specification; it can be the specification of a portion of your program (usually the most complex or the most critical). It has been proved countless times that the mere fact of writing S clarifies ideas, pops up unseen problems, etc. Is this a waste of time and resources, or an added cost? No, it is not. These unseen problems will flow into code and sooner or later they will produce unwanted behaviors. Actually, writing S can save you time and money.

Depending on the particular FM you are using, once you have S you can keep applying formal techniques such as:

1. S can be checked for different levels of soundness.
2. S can be used as a prototype.
3. S can be used to generate test cases.
4. S can be handled to the development team (i.e P is implemented from S).
5. S can be used to formally generate a correct-by-construction P.

All these activities improve the confidence on the correctness (either functional or security) of your software; only the last one can guarantee correctness but it is the most demanding in terms of time, effort and technical proficiency. Depending on the context, many of these techniques can be automated to some extent, and some times they can be fully automated. The first activity can uncover errors in S thus removing them before they make into P. The difference between 4 and 5 is that in 4 developers just read S and write P while in 5 developers *formally prove* that P implements S.

There are a number of FM and formal techniques. There are old, enduring, established, assessed, supported FM and there are new, experimental ones. None of them is a silver bullet: if it is very expressive, then it will be harder to automate, and vice versa; if its models are abstract, then program correctness will be harder to be formally proved; if its models are less abstract, then their complexity can be similar to that of the implementation. In general, each FM was thought and designed to be applied to a class of systems or for a particular verification activity. For example, there are FM to specify and analyze information systems; others more suitable for concurrency; there are very expressive FM aimed at full formal verification; and others aimed at automated verification of limited classes of programs.

In any case, the determining factor of a FM is its specification language. The specification language somewhat fixes what can be expressed, what formal techniques are possible and which can be automated to some valuable extent. The specification language determines the form of your models or specifications. Each specification language is based on some well-known mathematical or logic theory; they always have

a formal semantics. There are specification languages based on set theory, type theory, relation algebra, process algebra, transition systems, first-order logic, category theory, higher-order logic, temporal logic, etc. Some specification languages resemble high-level programming languages, most look like math+logic. Nevertheless, specification languages are to high-level programming languages what the latter are to machine code. That is, you can implement a formal specification in any given programming language.

2.2 Components of cryptocurrency software

A cryptocurrency software may encompass the following: cryptographic primitives, cryptographic protocols, consensus protocols, a blockchain, a virtual machine, a scripting language, a contract-oriented programming language, a wallet a number of smart contracts. Orthogonal to those the system runs on top of one or more operating systems and the TCP/IP stack, and it has been programmed in one or more programming languages with their corresponding compilers or interpreters. Yet orthogonal to all that you have complex functional and security requirements. And finally, a cryptocurrency software is usually a distributed system.

Any error on any of those components can potentially produce a failure or a vulnerability. That is, an error on any of those components can potentially affect the functional correctness of the system or its security.

Guaranteeing that a cryptocurrency software will never fail and will never be hacked (at least by using any known attack technique) entails proving that all those components are functional and security correct, which in turn requires a formal specification of each component and the interactions between them—which, in particular, includes giving a formal semantics to all the programming languages involved. If you program your blockchain on Rust and the Rust compiler has some error, then your system is potentially vulnerable; if you are using a proven cryptographic primitive whose implementation happens to hide an obscure error, then your system is potentially vulnerable.

Certifying the functional and security correctness of such a system is, currently, a challenging task even for seasoned FM experts. Fortunately, in many cases each component can be independently certified and, furthermore, many verification tasks (cf. items 1-4 above) can be carried on before attempting a formal correctness proof. Indeed, so far, the FM community has formally verified (some times parts of) some of the components of a cryptocurrency software such as, cryptographic primitives [2], compilers [3], communication protocols [4], cryptographic protocols [5], secure operating systems [6]. However, as far as we know, a formal proof of the scale and scope necessary for a full-fledged cryptocurrency software has never been attempted.

2.3 Cryptocurrency software as critical systems — Related work

If society will eventually depend on a cryptocurrency software its correctness proof should be attempted. Indeed, if a cryptocurrency software ever plays an important role in the society it will be that of a critical system. That is, of a system whose failure would cause irreparable loses—as a failure in the software controlling a nuclear power reactor. If the error or the attack causes the unwanted transfer of cryptomoney, the victims will not be able to recover it due to the virtual nature of cryptocurrencies and because many of these systems enforce strong privacy and anonymity protections.

On the other hand, critical software has been the main target of FM since their inception. For example, the railway industry has been applying FM for its critical systems for quite a long time [7]; computer security has been one of the traditional application domains of FM for almost 50 years [8, 9, 6, 10, 11, 12, 13]; several other industrial sectors such as nuclear, defense, health care, etc. apply FM to some of their critical software projects where, in average, 80% report no increase in development time, +90% report no added costs, and 88% report on improved quality [14].

Developers of cryptocurrency software should not be scared about using mathematics as a tool to describe software. In fact, Nakamoto uses math in his seminal paper on Bitcoin [15] and Wood uses it to describe the EVM [16]. However, these descriptions would not be understood as FM because they are not based on standardized notations nor on clear mathematical theories. On the other hand, recently, the FM community has started to pay attention to cryptocurrency software. Idelberger et al. [17] propose to use defeasible logic frameworks such as Formal Contract Logic for the description of smart contracts. Bhargavan et al. [18] compile Solidity programs into a verification-oriented functional language where they can verify source code. Luu et al. [19] use the Oyente tool to find and detect vulnerabilities in smart contracts. Hirai [20] use Lem to formally specify the EVM; Grishchenko, Maffei and Schneidewind [21] also formalize the EVM but in F*; and Hildenbrandt et al. do the same but with the reachability logic system known as $\mathbb{K}$. Pîrlea and Sergey [22] present a Coq [23, 24] formalization of a blockchain consensus protocol where some properties are formally verified.

More recently, Rosu [25] presented academic and commercial results in developing blockchain languages and virtual machines that come directly equipped with formal analysis and verification tools. Hajdu et

al. [26] developed a source-level approach for the formal specification and verification of Solidity contracts with the primary focus on events. Santos Reis et al. [27] introduced Tezla, an intermediate representation of Michelson smart contracts that eases the design of static smart contract analysers. In [28], Boyd et al. presented a blockchain model in Tamarin, that is useful for analyzing certain blockchain based protocols. On the other hand, Garfatta et al. [29] described a general overview of the different axes investigated actually by researchers towards the (formal) verification of Solidity smart contracts.

Additionally, Metere and Dong [30] present a mechanized formal verification of the Pedersen commitment protocol using EasyCrypt [31] and Fuchsbaue et al. [32] introduce an abstraction for the analysis of some security properties of the MimbleWimble cryptocurrency protocol [33, 34]. Finally, Betarte et al. [35, 36, 37] present formal properties and outline the basis of a model-driven verification approach to address the certification of the correctness of implementations of MimbleWimble.

2.4 A gradual adoption process

Hence, the cryptocurrency community should consider the adoption of FM for the developtment of its critical software components. However, the adoption of FM is not only a technical challenge but, more importantly, it requires a sort of cultural shift. In effect, the introduction of FM into the development process requires new personnel or a thorough training of existing staff; the introduction of new tools, techniques and processes; and, in some cases, a redesign of the development cycle—e.g., you cannot change source code without first (re)proving a theorem. Note that the works cited above are carried out by experts on FM. If the cryptocurrency community wants to use FM then it either has to establish strong, mid-term collaboration projects with academic groups or start an adoption program.

Due to the nature and dynamics of the cryptocurency ecosystem we think that developers should slowly introduce FM into the development of cryptocurrency software. FM should become a new asset of the community rather than an outsourced service. The following are some guidelines that can be considered:

1. Do not attempt a full formal correctness proof from the beginning.
2. Start with abstract specification languages (e.g. set theory) based on first-order logic.
3. Learn to formally specify.
4. Put the verification of code aside for later, focus on writing good models.
5. Abstract away cryptography, assume it is correct, focus on functional specifications.
6. Add *lightweight* or *automated* verification techniques such as simulation [38], prototyping [39], model-based testing [40], model-checking [41] and automated proofs [42, 43].
7. Assess what you have learned, correct the course and move on.

In point 5 we suggest that cryptography should be abstracted away in this first adoption stage. This might be generalized to security properties. The formal verification of security properties is still a challenging issue (see Appendix A). Formal verification of some cryptographic primitives has only been achieved very recently [2]. Furthermore, formally proving non-trivial security properties of code might be an overwhelming task in terms of the effort required, especially compared with proving functional correctness. In addition, many implementation details are orthogonal to the security properties to be established. This implies that slight changes in the implementation technology might have devastating consequences as concerns the security of the implementation.

Therefore, in this paper we show excerpts of a set-based formal model of two components of a cryptocurrency software: a consensus protocol (Sect. 3) and the Ethereum Virtual Machine (EVM, Sec. 4), where cryptography and security are abstracted away. Besides, we show how these models can be easily, although partially, analyzed by either lightweight or automated verification techniques (Sect. 5).

3 Formal Specification of a Consensus Protocol

There are several FM based on set theory and first-order logic (e.g. Z [44], VDM [45], B [46]). These notations are designed to write large, complex specifications; and there is a number of tools to work with them such as editors, typecheckers, theorem provers, animators, etc. However, in this paper we will use the plain and simple language of mathematics extended with a couple of conventions, to avoid explaining the peculiarities of a particular notation. This will be enough as to show the 'look-and-feel' of such specifications.

The following are some snippets of a model of a consensus protocol based on the work by Pîrlea and Sergey [22].

A consensus protocol deals with addresses, hashes, proof objects, etc. In an abstract model of the protocol the internal structure of these entities is irrelevant. In particular, hashes and proof objects are strongly related to cryptography which is a feature we think should be abstracted away at this point. Then, we have the set of all possible addresses ($Addr$) that can be used. If $a \in Addr$ then a is an address of the protocol. We do not know what a's structure is, how it was generated, whether it is 128 bits or 256 bits long, etc. The idea is that we do not need to know those things now, because they are implementation details which will not alter the fundamental behavior of the protocol. In this sense we also have the set of hashes ($Hash$), the set of proofs objects ($Proof$) and the set of transactions (Tx). The only condition required for these sets is that they come equipped with equality and be pairwise disjoint.

$$[[Addr, Hash, Proof, Tx]]$$

The block data structure (cf. blockchain) is a record with three fields: $prev$, (usually) points to the parent block; txs, stores the sequence of transactions stored in the block; and pf is a proof object required to validate the block. Then we define $Block$ as the set of all such records:

$$Block \mathrel{\widehat=} [prev : Hash;\ txs : \mathrm{seq}\ Tx;\ pf : Proof]$$

The local state space of a participating network node is given by three state variables: as, are the addresses of the peers this node is aware of; bf, is a block forest (not shown) which records the minted and received blocks; and tp, is a set of received transactions which eventually will be included in minted blocks.

$$LocState \mathrel{\widehat=} [as : \mathbb{P}\ Addr;\ bf : Hash \pfun Block;\ tp : \mathbb{P}\ Tx]$$

The protocol configuration is represented by two state variables: $Delta$, which establishes a mapping between network addresses and the corresponding node (local) states (in [22] this variable is referred to as the *global state*); and P, a set of packets (which represent the messages exchanged by nodes).

$$Conf \mathrel{\widehat=} [Delta : Addr \pfun LocState;\ P : \mathbb{P}\ Packet]$$

Packets are just tuples of two addresses (origin and destination) and a message.

$$Packet == Addr \times Addr \times Msg$$

The model has twelve state transitions divided into two groups: *local* and *global*. Local transitions are those executed by network nodes, while global transitions promote local transitions to the network level. In turn, the local transitions are grouped into *receiving* and *internal* transitions. Receiving transitions model the nodes receiving messages from other nodes and, possibly, sending out new messages; internal transitions model the execution of instructions run by each node when some local condition is met. Here, we show only the local, receiving transition named $RcvAddr$.

$$\begin{array}{l} RcvAddr(s : LocState;\ p? : Packet;\ ps! : \mathbb{P}\ Packet; \\ \quad s' : LocState) \mathrel{\widehat=} \\ \qquad p?.2 = this \land \\ \qquad \exists\, asm : \mathbb{P}\ Addr \bullet \\ \qquad\quad p?.3 = AddrMsg\ asm \land \\ \qquad\quad s'.as = s.as \cup asm \land \\ \qquad\quad s'.bf = s.bf \land s'.tp = s.tp \land \\ \qquad\quad ps! = \{a : asm \setminus as \bullet (p?.2, a, ConnectMsg)\} \\ \qquad\qquad \cup \{a : as \bullet (p?.2, a, AddrMsg\ p'.as)\} \end{array}$$

As can be seen, $RcvAddr$ transitions from the local state s into a new local state s', receives a packet ($p?$), and sends out a set of packets ($ps!$). The node checks whether or not the packet's destination address coincides with its own address. In that case, the node adds the received addresses to its local state and sends out a set of packets that are either of the form $(p?.2, a, ConnectMsg)$ or $(p?.2, a, AddrMsg\ p'.as)$. The former are packets generated from the received addresses and sent to the new peers the node now knows, while the latter are messages telling its already known peers that it has learned of new peers.

Observe that the expression after the $\mathrel{\widehat=}$ symbol in $RcvAddr$ is a predicate. That is, for example, the equal symbol in $s'.as = s.as \cup asm$ represents logical equality and not (imperative) assignment; $\land$ means logical conjunction and thus $p \land q$ is equal to $q \land p$; $\cup$ is set union; and so forth.

Hopefully, this simple example lets the reader see that abstract specifications tend to be much simpler, concise and evidently correct than code. Furthermore, much of what is said in an abstract specification is what seasoned programmers think while designing good code. Then, formal specification languages give programmers an efficient technique by means of which they can write down their best ideas and be able to communicate them easily and unambiguously. In most cases using formal specifications in this way provides costs savings rather than the opposite.

4 Formal Specification of the EVM

The same notation and methodology can be used to formally specify the EVM. In this case we depart from the Yellow Paper [16]. The state of the EVM is given by three records:

$$EVMState \mathrel{\widehat=} [World;\ Machine;\ CallStack]$$

where, for example *World* and *Machine* are defined as follows:

$$\begin{aligned} World \mathrel{\widehat=} & [acc, accCC : Addr \pfun Acc; \\ & newaddr : Addr;\ step : STEP] \\ Machine \mathrel{\widehat=} & [g : ETH;\ pc : \mathbb{N};\ m : A \pfun B;\ i : \mathbb{N}; \\ & s : \mathrm{seq}\, W;\ out : P] \end{aligned}$$

where in turn, for instance, *acc* represents the Ethereum accounts as a partial function from the set of addresses onto the set of records $Acc \mathrel{\widehat=} [nonce : \mathbb{N};\ bal : ETH;\ code : PROG]$, which stores the main information of the accounts; *step* records the current transaction execution step; m represents the state of the memory and g the available gas—where A, B, W, etc. are given sets like $Addr$.

We have two types of transactions:

$$TT \mathrel{\widehat=} \{contractCreation, messageCall\}$$

A transaction can be modeled as a record with several fields (some are omited):

$$\begin{aligned} Transaction \mathrel{\widehat=} & [Tn, Tg : \mathbb{N};\ Tp, Tv : ETH;\ Ti : P; \\ & Td : \mathrm{seq}\, B;\ snd : Addr] \end{aligned}$$

where, for instance, Tn is the nonce, Tg is the gas limit, Tp is the gas price and Ti is the initialization program for the account.

Now we can model how transactions are executed. According to the YP, a transaction is executed in four steps: *(a)* the checkpoint state ($\boldsymbol{\sigma}_0$, in YP's notation); *(b)* the post-execution provisional state ($\boldsymbol{\sigma}_P$); *(c)* the pre-final state ($\boldsymbol{\sigma}^*$); and *(d)* the final state ($\boldsymbol{\sigma}'$) is reached after deleting all accounts that either appear in the self-destruct list or are touched and empty. The specification of the processing step called checkpoint state ($\boldsymbol{\sigma}_0$) is independent of the transaction type and is defined as follows:

$$\begin{aligned} & CheckpointState(s : World;\ t? : Transaction; \\ & \quad s' : World) \mathrel{\widehat=} \\ & \qquad TransactionValidity(s, t?) \land \\ & \qquad UpdateSender(s.acc\ t?.sender, \\ & \qquad\quad (acc\ t?.sender).bal, t?.Tp, t?.Tg, a') \land \\ & \qquad s.step = initial \land \\ & \qquad s'.acc = s.acc \oplus \{(t?.sender, a')\} \land \\ & \qquad s'.step = ccbegins \end{aligned}$$

where *TransactionValidity* (not shown) is a complex predicate stating the conditions for a transaction to be valid and *UpdateSender* is defined as follows:

$$\begin{aligned} & UpdateSender(s : Acc;\ b, p : ETH;\ g : \mathbb{N};\ s' : Acc) \mathrel{\widehat=} \\ & \quad s'.bal = b - g * p \land s'.nonce = s.nonce + 1 \land \\ & \quad s'.code = s.code \end{aligned}$$

Hence, *CheckpointState* checks whether the requested transaction is valid and in that case it (only) updates the sender account by debiting the result of multiplying the gas price and the gas limit as given in the transaction and by adding one to the number of transactions ($s.nonce$).

The transition from $\boldsymbol{\sigma}_0$ to $\boldsymbol{\sigma}_P$ requires the execution of the program associated to the account (or the initialization program if it is a contract creation). Then, at this point we should specify how the EVM executes (bytecode) programs. To this end, we first need to formalize each and every EVM bytecode instruction. Most instructions can be modeled as a transition between two states. We will do that only for a simplified version of the instruction called `create`. According to the YP `create` "Creates a new account with associated code". We have identified two cases so we have:

$$Create \mathrel{\widehat=} Create1 \lor Create2$$

where *Create*1 is not shown due to space restrictions. *Create*2 formalizes a situation where the account is actually not created due to some boundary condition:

$$
\begin{array}{l}
\mathit{Create}2(q : \mathit{Machine};\ w : \mathit{World};\ a? : \mathit{Addr};\ n? : \mathbb{N};\\
\quad q' : \mathit{Machine}) \mathrel{\widehat=}\\
\qquad q.s\,1 > (w.\mathit{acc}\ a?).\mathit{bal} \lor n? \geq 1024 \land\\
\qquad q'.s = \langle 0 \rangle \frown \mathit{tail}(\mathit{tail}(\mathit{tail}\ q.s)) \land\\
\qquad q'.i = M(q.i, q.s\,2, q.s\,3) \land\\
\qquad q'.m = q.m \land q'.g = q.g
\end{array}
$$

where M updates the amount of used memory ($q.i$) by using the second and third positions of the machine stack ($q.s$). Note that *Create*2 modifies the machine state but it needs to access the world state. It also receives the address of the account which owns the code that is executing ($a?$) and the number of *Creates* being executed at present ($n?$). The account is not created because either the balance of the caller is too low to fulfill the value transfer or there are too many active *Create* calls ($n? \geq 1024$). In this case the first three positions in the stack are removed and a 0 is stacked on top of that.

Lesson learned. The YP describes the semantics of the EVM mixing informal text with formulas written in some ad-hoc mathematical notation. In the process of writing this specification we found many obscure (probably inconsistent) issues in the math used in the YP. Established formal specification languages have gone a long process of standardization and analysis; their fundamentals have been studied for decades (maybe centuries) by some of the founding fathers of modern mathematics and Computer Science. Even the math used in the YP cannot be compared to well established FM. Besides, most FM are supported by tools implementing a variety of verification techniques. Using math in the YP is a good step forward but the invested effort would be at least partially wasted.

5 Some Verification Techniques for Set-Based Models

In this section we show the application of three verification techniques that can be used when set-based formal specifications are available. In the first one we show how a set-based specification can be turned into a set-based prototype, that is an inefficient program that nonetheless is correct-by-construction and so it can be used to analyze complex situations (Sect. 5.1). Next, we show how the same set-based model can be used to generate test cases that can be used to test the implementation (Sect. 5.2). Finally, we automatically prove that the same models enjoy some properties (Sect. 5.3).

5.1 Set-based prototypes and simulations

Set-based specifications such as those presented in sections 3 and 4, can be easily turned into set-based prototype programs. In effect, a set-based model can be encoded in the programming language provided by the {*log*} ('setlog') tool [47, 48]. {*log*} is a programming language, a satisfiability solver and an automated theorem prover where sets are first-class entities. {*log*} provides the usual Boolean connectives and most of the set and relational operators available in set theory including binary relations. Hence, it is quite natural to encode a set-based specification as a {*log*} program. Given that {*log*} is based on Prolog its programs resemble Prolog programs.

The {*log*} encoding of *RcvAddr* presented in Sect. 3 is the following:

```
rcvAddr(S,P,Ps,S_) :-
  S = {[as,As] / Rest} &
  P = [_,this, addrMsg(Asm)] &
  un(As,Asm,As_) &
  diff(Asm,As,D) &
  PsD = ris(A in D,[],true,[this,A,
      connectMsg]) &
  PsAs = ris(A in As,[],true,[this,A,
      addrMsg(As_)]) &
  un(PsD,PsAs,Ps) &
  S_ = {[as,As_] / Rest}.
```

`rcvAddr` is a {*log*} clause respecting the interface of *RcvAddr*. In this case, however, instead of using set membership to somewhat "type" the parameters we rest on unification. As in Prolog, {*log*} programs are based on unification with the addition of set unification. Variables must start with an uppercase letter. We

take advantage of set theory to encode other data structures such as records. A record is a set of ordered pairs where the first component names the field and the second is the variable holding the values. Hence, a statement such as `S = {[as,As] / Rest}` (set) unifies the first parameter with a set term singling out the record field needed in this case (`As`) and the rest of the record (`Rest`). The same is done with packet `P` where `_` means any value as first component and `addrMsg(Asm)` gets the set of addresses received in the packet without introducing an existential quantifier. The set comprehensions used in the specification are implemented with {*log*}'s so-called Restricted Intentional Sets (RIS) [49]. A RIS is interpreted as a set comprehension where the control variable ranges over a finite set (`D` and `As`). Finally, `diff(A,B,C)` is interpreted as $C = A \setminus B$ and `un(A,B,C)` as $C = A \cup B$.

A clause such as `rcvAddr` can be seen both as a {*log*} formula and as a {*log*} program. It is a formula in the sense that `&` is logical conjunction and so the order of statements in `rcvAddr` is irrelevant due to commutativity. It is a program simply because we can use the {*log*} interpreter to compute outputs from inputs. However, given that {*log*} programs are meant to be prototypes we talk of *simulations* or *animations* rather than executions.

Then, given that `S` and `P` are meant to be inputs while `Ps` and `S_` are outputs, we can run a simulation from an initial *LocState* and input packet $p?$ such as:

```
S = {[as,{}] / _}
P = [_,this,addrMsg({a1,a2})]
```

Now, for instance, we can call `rcvAddr` twice chaining before and after states as follows:

```
S = {[as,{}] / _} &
P = [_,this,addrMsg({a1,a2})] &
rcvAddr(S,P,Ps1,S1) &
rcvAddr(S1,[_,this,addrMsg({a1,a3})],
    Ps2,S2).
```

in which case {*log*} returns:

```
Ps1 = ris(A in {a1,a2/_N2},[],true,
    [this,A,connectMsg],true),
S1  = {[as,{a1,a2}]/R},
Ps2 = {[this,a3,connectMsg],[this,a1,
    addrMsg({a2,a1,a3})],
    [this,a2,addrMsg({a2,a1,a3})] /
    ris(A in _N1,[],true,[this,A,
    connectMsg],true)},
S2  = {[as,{a2,a1,a3}]/R}
Constraint: subset(_N2,{a1,a2}),
    subset(_N1,{a1,a3}), a1 nin _N1,
    a2 nin _N1
```

That is, {*log*} binds values for all the free variables in a way that the formula is satisfied (if it is satisfiable at all). In this way we can trace the execution of the protocol w.r.t. states and outputs by starting from a given state (e.g. `S`) and input values (e.g. `[_,this,addrMsg({a1,a2})]`), and chaining states throughout the execution of the state transitions included in the simulation (e.g. `S1` and `S2`).

5.2 Model-based testing

Model-based testing (MBT) is a testing methodology where test cases are drawn from models or program specifications [40]. That is, instead of letting testers to think what test cases are necessary to test a program, a MBT method gives a disciplined, systematic and quantifiable algorithm for test case generation. There are a number of MBT methods depending on the FM and the type of systems under consideration [50]. Most MBT methods are supported by (semi)automatic tools thus turning program testing into a more efficient process.

In particular, the Test Template Framework (TTF) can be applied to set-based specifications [51]. Then, this means that, by writing a set-based specification you can get, almost for free, a prototype and test cases to test the implementation.

We will apply a reduced version of the TTF to the *CheckpointState* specification given in Sect. 4—see elsewhere in the literature for more examples and applications to critical software [52, 53]. Note that in *CheckpointState* we have the following post-condition:

$$s'.acc = s.acc \oplus \{(t?.sender, a')\} \tag{1}$$

where $\oplus$ is a relational operator stating that if $t?.sender \in \operatorname{dom} s.acc$ then its relational image must be updated with a', otherwise the pair $(t?.sender, a')$ must be added to $s.acc$. Besides, recall that acc is a partial function, i.e. a binary relation where all first components are different from each other. Such data structures and operators are seldom available in high-level programming languages. Hence, an efficient implementation of a predicate like (1) will yield a non-trivial piece of code that deserves to be thoroughly tested, specially when it belongs to a critical system.

The TTF defines so-called *standard partitions* for each set theoretic operator. The standard partition for $R \oplus G$ is the following:

$$
\begin{array}{ll}
R = \emptyset, G = \emptyset & R \neq \emptyset, G \neq \emptyset, \operatorname{dom} G \subset \operatorname{dom} R \\
R = \emptyset, G \neq \emptyset & R \neq \emptyset, G \neq \emptyset, \operatorname{dom} R \cap \operatorname{dom} G = \emptyset \\
R \neq \emptyset, G = \emptyset & R \neq \emptyset, G \neq \emptyset, \operatorname{dom} R \subset \operatorname{dom} G \\
R \neq \emptyset, G \neq \emptyset, & R \neq \emptyset, G \neq \emptyset, \operatorname{dom} R \cap \operatorname{dom} G \neq \emptyset, \\
\quad \operatorname{dom} R = \operatorname{dom} G & \quad \neg (\operatorname{dom} G \subseteq \operatorname{dom} R), \\
 & \quad \neg (\operatorname{dom} R \subseteq \operatorname{dom} G)
\end{array}
$$

This means that an expression of the form $R \oplus G$ should be tested with eight test cases as follows: R and G equal to the empty set; R equal to the empty set and G not equal to the empty set and vice versa; R and G different from the empty set but having the same domain; and so forth. If for a particular expression some combination of R and G is unfeasible, then it is simply discarded.

Thus, when the standard partition for $\oplus$ is applied to (1) in the context of *CheckpointState* it yields, after some simplifications, only two test conditions:

$$
\begin{array}{l}
\operatorname{dom} s.acc = \{t?.sender\} \\
\{t?.sender\} \subset \operatorname{dom} s.acc
\end{array}
$$

That is, it make sense to test the implementation of (1) in *CheckpointState* when: the address of the transaction sender is the only account in the system and when it is not the only account in the system. However, there is more to be tested in *CheckpointState*. For example, *TransactionValidity* is a complex predicate where the TTF can be applied in several ways. Furthermore, the TTF dictates how the test conditions for some lines of code must be combined with the test conditions generated for the other lines of code.

In general, all the steps of the TTF can be (semi)automated by tools such as FASTEST [53] and other MBT methods for set-based specifications provide good tool support as well [54].

5.3 Automated proofs

A program is correct if it verifies its specification. But if the specification is wrong, the program will be wrong and this will be undetectable. Therefore, effort must be made to ensure the specification is correct. The confidence on the correctness of the specification can be increased by proving that it enjoys many desired properties. In a full formal context, all these proofs should be mechanized, although a good first approximation are manual proofs and a second best are automated proofs, as in general full automation is impossible.

For set-based specifications {*log*} can be used as an automated theorem prover and as a counterexample generator—that is, if a proof fails we can know why that happened. In this case we see {*log*} code as formulas over the theory of finite sets (cf. Sect. 5.1). In order to prove that a {*log*} formula is a theorem we actually need to prove that its negation is unsatisfiable. For example, concerning `rcvAddr`, we might prove that `PsD` and `PsAs` are disjoint sets. In this case we can submit the following to {*log*}:

```
diff(Asm,As,D) &
PsD = ris(A in D,[],true,
     [this,A,connectMsg]) &
PsAs = ris(A in As,[],true,[this,A,
     addrMsg(As_)]) &
ndisj(PsD,PsAs).
```

where `ndisj(A,B)` is interpreted as $A \cap B \neq \emptyset$. In this case {*log*} returns `no` which means that the formula is unsatisfiable and so `PsD` and `PsAs` are disjoint.

As another example, we may want to prove that $s.acc$ is still a partial function after executing *CheckpointState*. This is a so-called *state invariant preservation theorem*. These theorems are of the following form:

$$Inv \wedge Op \Rightarrow Inv' \tag{2}$$

that is, if *Inv* holds before executing *Op*, then *Inv* also holds in the after state (i.e., Inv' is true, where $Inv' \mathrel{\widehat{=}} Inv[\forall\, v, v'/v]$). If we want to use $\{log\}$ to discharge such a theorem we must check whether the negation of (2) is unsatisfiable. Then, in this case the formula to be submitted to $\{log\}$ is:

```
World = {[acc,Acc] / Rest} &
pfun(Acc) &
checkpointState(World,Trans,World_) &
World_ = {[acc,Acc_] / Rest} &
npfun(Acc_).
```

where `pfun(F)` is interpreted as F is a partial function and `npfun` as its negation. In this case $\{log\}$ returns `no`, which means that `checkpointState` preserves the invariant, as otherwise expected.

6 Final remarks

Cryptocurrency software should be considered critical: its failures and vulnerabilities would cause irreparable loses. Formal methods have proved to be successful in delivering bug-free software for a range of critical systems. Hence, our first conclusion is that the cryptocurrency community should pay attention to formal methods. However, integrating formal methods into the development process of highly innovative domains cannot be done all at once. Then, our second conclusion is that the cryptocurrency community should *gradually* adopt formal methods.

Specifically, we propose to start the adoption process by using formal specification languages based on set theory and first-order logic. Most developers have been exposed to the mathematics underlying these languages, so adoption could be made minimizing the learning curve. Set-based specifications accurately and concisely describe cryptocurrency software. In order to support this claim we formally specified representative parts of two key components of cryptocurrency software. Once a set-based specification has been written, several verification techniques are enabled. In particular we have shown how prototypes can be easily generated and simulations can be run on them; then, we have applied a model-based testing method to generate test cases from a set-based specification; and finally, we have automatically discharged some proof obligations.

However, these techniques do not allow the formal verification of the implementation. To this end more powerful languages and techniques (e.g. the `Coq` proof assistant [23, 24]) can and should be adopted in future stages. This should be done once the use of formal methods is well understood by the community. Hopefully, the contents of this paper will convince key players of the cryptocurrency community to assess the application of formal methods more thoroughly.

Finally, in [36] we have highlighted elements that constitute essential steps towards the development of an exhaustive formalization (using state machines) of the MimbleWimble cryptocurrency protocol, the analysis of its (security) properties and the verification of its implementations (*Grin* and *Beam*), following approaches outlined in this paper. We plan to continue working on these lines, also considering tools oriented towards the verification of cryptographic protocols and implementations, such as `EasyCrypt` [31], `ProVerif` [55], and `CryptoVerif` [56].

References

[1] V. Buterin, "Critical update re: Dao vulnerability, 2017," Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability. Last access: September 2021.

[2] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, T. Ramananandro, A. Rastogi, N. Swamy, C. Wintersteiger, and S. Z. Béguelin, "Evercrypt: A fast, verified, cross-platform cryptographic provider," *IACR Cryptology ePrint Archive*, vol. 2019, p. 757, 2019. [Online]. Available: https://eprint.iacr.org/2019/757

[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009. [Online]. Available: https://doi.org/10.1145/1538788.1538814

[4] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, R. T. Morris and S. Savage, Eds. USENIX, 2004, pp. 155–168. [Online]. Available: http://www.usenix.org/events/nsdi04/tech/musuvathi.html

[5] G. Barthe, D. Hedin, S. Z. Béguelin, B. Grégoire, and S. Heraud, "A machine-checked formalization of sigma-protocols," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*. IEEE Computer Society, 2010, pp. 246–260. [Online]. Available: https://doi.org/10.1109/CSF.2010.24

[6] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *ACM Trans. Comput. Syst.*, vol. 32, no. 1, pp. 2:1–2:70, 2014. [Online]. Available: https://doi.org/10.1145/2560537

[7] T. Lecomte, D. Déharbe, É. Prun, and E. Mottin, "Applying a formal method in industry: A 25-year trajectory," in *Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, ser. Lecture Notes in Computer Science, S. A. da Costa Cavalheiro and J. L. Fiadeiro, Eds., vol. 10623. Springer, 2017, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-319-70848-5_6

[8] L. LaPadula, D. E. Bell, and L. J. LaPadula, "Secure computer systems: Mathematical foundations, draft mtr, the mitre corporation, 1973," Available: http://www-personal.umich.edu/~cja/LPS12b/refs/belllapadula1.pdf. Last access: September 2021.

[9] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. [Online]. Available: https://doi.org/10.1109/SP.1982.10014

[10] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "System-level non-interference of constant-time cryptography. part I: model," *J. Autom. Reasoning*, vol. 63, no. 1, pp. 1–51, 2019. [Online]. Available: https://doi.org/10.1007/s10817-017-9441-5

[11] G. Betarte, J. D. Campo, C. Luna, and A. Romano, "Formal analysis of android's permission-based security model,," *Sci. Ann. Comp. Sci.*, vol. 26, no. 1, pp. 27–68, 2016. [Online]. Available: http://dx.doi.org/10.7561/SACS.2016.1.27

[12] G. Barthe, G. Betarte, J. D. Campo, C. D. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1267–1279. [Online]. Available: https://doi.org/10.1145/2660267.2660283

[13] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie, "System-level non-interference of constant-time cryptography. part II: verified static analysis and stealth memory," *J. Autom. Reason.*, vol. 64, no. 8, pp. 1685–1729, 2020. [Online]. Available: https://doi.org/10.1007/s10817-020-09548-x

[14] J. S. Fitzgerald, J. Bicarregui, P. G. Larsen, and J. Woodcock, "Industrial deployment of formal methods: Trends and challenges," in *Industrial Deployment of System Engineering Methods*, A. B. Romanovsky and M. Thomas, Eds. Springer, 2013, pp. 123–143. [Online]. Available: https://doi.org/10.1007/978-3-642-33170-1_10

[15] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, 2008." Available: https://bitcoin.org/bitcoin.pdf. Last access: September 2021.

[16] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger eip-150 revision (759dccd - 2017-08-07)," 2017, Last access: September 2021. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[17] F. Idelberger, G. Governatori, R. Riveret, and G. Sartor, "Evaluation of logic-based smart contracts for blockchain systems," in *Rule Technologies. Research, Tools, and Applications - 10th International Symposium, RuleML 2016, Stony Brook, NY, USA, July 6-9, 2016. Proceedings*, ser. Lecture Notes in Computer Science, J. J. Alferes, L. E. Bertossi, G. Governatori, P. Fodor, and D. Roman, Eds., vol. 9718. Springer, 2016, pp. 167–183. [Online]. Available: https://doi.org/10.1007/978-3-319-42019-6_11

[18] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*, T. C. Murray and D. Stefan, Eds. ACM, 2016, pp. 91–96. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993611

[19] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[20] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds., vol. 10323. Springer, 2017, pp. 520–535. [Online]. Available: https://doi.org/10.1007/978-3-319-70278-0_33

[21] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, ser. Lecture Notes in Computer Science, L. Bauer and R. Küsters, Eds., vol. 10804. Springer, 2018, pp. 243–269. [Online]. Available: https://doi.org/10.1007/978-3-319-89722-6_10

[22] G. Pîrlea and I. Sergey, "Mechanising blockchain consensus," in *Proc. of CPP 2018*. New York, USA: ACM, 2018, pp. 78–90. [Online]. Available: http://doi.acm.org/10.1145/3167086

[23] The Coq Team, "The Coq proof assistant reference manual," Available: http://coq.inria.fr. Last access: September 2021.

[24] Y. Bertot, P. Castran, G. i. Huet, and C. Paulin-Mohring, *Interactive theorem proving and program development: Coq'Art : the calculus of inductive constructions*, ser. Texts in theoretical computer science. Berlin, New York: Springer, 2004, donnes complmentaires http://coq.inria.fr. [Online]. Available: http://opac.inria.fr/record=b1101046

[25] G. Rosu, "Formal Design, Implementation and Verification of Blockchain Languages Using K (Invited Talk)," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, ser. OpenAccess Series in Informatics (OASIcs), B. Bernardo and D. Marmsoler, Eds., vol. 84. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 1:1–1:1. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13414

[26] Á. Hajdu, D. Jovanovic, and G. F. Ciocarlie, "Formal specification and verification of solidity contracts with events (short paper)," in *2nd Workshop on Formal Methods for Blockchains, FMBC@CAV 2020, July 20-21, 2020, Los Angeles, California, USA (Virtual Conference)*, ser. OASIcs, B. Bernardo and D. Marmsoler, Eds., vol. 84. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 2:1–2:9. [Online]. Available: https://doi.org/10.4230/OASIcs.FMBC.2020.2

[27] J. S. Reis, P. Crocker, and S. M. de Sousa, "Tezla, an Intermediate Representation for Static Analysis of Michelson Smart Contracts," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, ser. OpenAccess Series in Informatics (OASIcs), B. Bernardo and D. Marmsoler, Eds., vol. 84. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 4:1–4:12. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13417

[28] C. Boyd, K. Gjøsteen, and S. Wu, "A Blockchain Model in Tamarin and Formal Analysis of Hash Time Lock Contract," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*, ser. OpenAccess Series in Informatics (OASIcs), B. Bernardo and D. Marmsoler, Eds., vol. 84. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 5:1–5:13. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13418

[29] I. Garfatta, K. Klai, W. Gaaloul, and M. Graiet, "A survey on formal verification for solidity smart contracts," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437378.3437879

[30] R. Metere and C. Dong, "Automated cryptographic analysis of the pedersen commitment scheme," in *Computer Network Security - 7th International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2017, Warsaw, Poland, August 28-30, 2017, Proceedings*, ser. Lecture Notes in Computer Science, J. Rak, J. Bay, I. V. Kotenko, L. J. Popyack, V. A. Skormin, and K. Szczypiorski, Eds., vol. 10446. Springer, 2017, pp. 275–287. [Online]. Available: https://doi.org/10.1007/978-3-319-65127-9_22

[31] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P. Strub, "Easycrypt: A tutorial," in *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, ser. LNCS, A. Aldini, J. López, and F. Martinelli, Eds., vol. 8604. Springer, 2013, pp. 146–166. [Online]. Available: https://doi.org/10.1007/978-3-319-10082-1_6

[32] G. Fuchsbauer, M. Orrù, and Y. Seurin, "Aggregate cash systems: A cryptographic investigation of mimblewimble," in *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, Y. Ishai and V. Rijmen, Eds., vol. 11476. Springer, 2019, pp. 657–689. [Online]. Available: https://doi.org/10.1007/978-3-030-17653-2_22

[33] T. Jedusor, "Mimblewimble, 2016," Available: https://scalingbitcoin.org/papers/mimblewimble.txt. Last access: September 2021.

[34] A. Poelstra, "Mimblewimble, 2016," Available: https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf. Last access: September 2021.

[35] G. Betarte, M. Cristiá, C. D. Luna, A. Silveira, and D. Zanarini, "Towards a formally verified implementation of the mimblewimble cryptocurrency protocol," in *Applied Cryptography and Network Security Workshops - ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19-22, 2020, Proceedings*, ser. Lecture Notes in Computer Science, J. Zhou, M. Conti, C. M. Ahmed, M. H. Au, L. Batina, Z. Li, J. Lin, E. Losiouk, B. Luo, S. Majumdar, W. Meng, M. Ochoa, S. Picek, G. Portokalidis, C. Wang, and K. Zhang, Eds., vol. 12418. Springer, 2020, pp. 3–23. [Online]. Available: https://doi.org/10.1007/978-3-030-61638-0_1

[36] A. Silveira, G. Betarte, M. Cristiá, and C. Luna, "A formal analysis of the MimbleWimble cryptocurrency protocol," *Sensors*, vol. 21, no. 17, p. 5951, 2021. [Online]. Available: https://doi.org/10.3390/s21175951

[37] G. Betarte, M. Cristiá, C. D. Luna, and A. Silveira, "A range proof scheme analysis for the mimblewimble cryptocurrency protocol," To be published in IEEE URUCON 2021 Conference (2021).

[38] S. Hallerstede, M. Leuschel, and D. Plagge, "Validation of formal models by refinement animation," *Sci. Comput. Program.*, vol. 78, no. 3, pp. 272–292, 2013. [Online]. Available: https://doi.org/10.1016/j.scico.2011.03.005

[39] M. Cristiá and G. Rossi, "Rapid prototyping and animation of Z specifications using {*log*}," in *1st International Workshop about Sets and Tools (SETS 2014)*, 2014, pp. 4–18, informal proceedings: http://sets2014.cnam.fr/papers/sets2014.pdf. Last access: September 2021.

[40] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007. [Online]. Available: http://www.elsevierdirect.com/product.jsp?isbn=9780123725011

[41] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 2001. [Online]. Available: http://books.google.de/books?id=Nmc4wEaLXFEC

[42] M. Newborn, *Automated theorem proving - theory and practice*. Springer, 2001. [Online]. Available: http://www.springer.com/computer/swe/book/978-0-387-95075-4

[43] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[44] J. M. Spivey, *The Z notation: a reference manual*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1992.

[45] C. B. Jones, *Systematic Software Development Using VDM (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[46] J.-R. Abrial, *The B-book: Assigning Programs to Meanings*. New York, NY, USA: Cambridge University Press, 1996.

[47] M. Cristiá and G. Rossi, "Solving quantifier-free first-order constraints over finite sets and binary relations," *J. Autom. Reasoning*, vol. 64, no. 2, pp. 295–330, 2020. [Online]. Available: https://doi.org/10.1007/s10817-019-09520-4

[48] G. Rossi. (2008) {*log*}. [Online]. Available: http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html

[49] M. Cristiá and G. Rossi, "Automated reasoning with restricted intensional sets," *J. Autom. Reason.*, vol. 65, no. 6, pp. 809–890, 2021. [Online]. Available: https://doi.org/10.1007/s10817-021-09589-w

[50] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 1–76, 2009.

[51] P. Stocks and D. A. Carrington, "A framework for specification-based testing," *IEEE Trans. Software Eng.*, vol. 22, no. 11, pp. 777–793, 1996. [Online]. Available: https://doi.org/10.1109/32.553698

[52] M. Cristiá, P. Albertengo, C. S. Frydman, B. Plüss, and P. R. Monetti, "Applying the test template framework to aerospace software," in *34th Annual IEEE Software Engineering Workshop, SEW 2011, Limerick, Ireland, June 20-21, 2011*, J. L. Rash and C. A. Rouff, Eds. IEEE Computer Society, 2011, pp. 128–137. [Online]. Available: https://doi.org/10.1109/SEW.2011.25

[53] ——, "Tool support for the test template framework," *Softw. Test., Verif. Reliab.*, vol. 24, no. 1, pp. 3–37, 2014. [Online]. Available: http://dx.doi.org/10.1002/stvr.1477

[54] M. Leuschel and M. J. Butler, "Prob: an automated analysis toolset for the B method," *STTT*, vol. 10, no. 2, pp. 185–203, 2008. [Online]. Available: https://doi.org/10.1007/s10009-007-0063-9

[55] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, ser. CSFW '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 82–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872752.873511

[56] ——, "CryptoVerif: A computationally sound mechanized prover for cryptographic protocols," in *Dagstuhl seminar "Formal Protocol Verification Applied"*, Oct. 2007. [Online]. Available: http://prosecco.gforge.inria.fr/personal/bblanche/publications/BlanchetDagstuhl07.html

[57] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, 1985. [Online]. Available: https://doi.org/10.1016/0020-0190(85)90056-0

[58] J. P. Anderson, "Computer Security technology planning study," http://csrc.nist.gov/publications/history/ande72.pdf, Deputy for Command and Management System, USA, Tech. Rep., 1972.

A Subtleties and Traps with Security Properties

Security properties have proved to be unexpectedly complex to grasp and formalize. One of the fundamental reasons for this is that security properties are not safety properties (w.r.t. the Alper-Schneider framework [57]). Most of the verification methods, results and tools developed for almost half a century by the Software Engineering community have been aimed at safety properties. Then, most of them fall short when dealing with security properties. Only very recently some of these methods and tools have been adapted for the problem of security properties.

For instance, safety properties enjoy, what is some times called, the *refinement property*. This means that, if S is the description of a system verifying safety property F and S' is a more deterministic refinement of S, then S' (automatically) verifies F. However, in general, security properties do not enjoy the refinement property. This means that even if you have proved that S verifies security property F and that S' is a refinement of S, then you cannot assert that S' verifies F. The problem is that there are implementation decisions that in spite of generating a more deterministic system, they might introduce security issues. A typical example is the specification of the values stored in the memory cells delivered by the operating system to a process. At a certain level of abstraction these values might be underspecified which implies that, potentially, any value can be stored. Assume you can prove that this is secure. Afterwards, programmers decide to deliver the cells with the values left by the last process that used that buffer. Although this is an implementation of the first specification it is blatantly insecure.

On the other hand, reasoning about implementations provides the ultimate guarantee that deployed mechanisms behave as expected. However, due to the issues mentioned above, formally proving non-trivial security properties of code might be an overwhelming task in terms of the effort required, especially compared w.r.t. proving functional correctness (i.e. a safety property). In addition, many implementation details are orthogonal to the security properties to be established. This implies that slight changes in the implementation

technology might have devastating consequences as concerns the security of the implementation. Therefore, complementary approaches are needed when non-trivial security properties are at stake:

i) Verification is performed on idealized models that abstract away the specifics of any particular implementation, and yet provide a realistic setting in which to explore the security issues that pertain to the realm of those (critical) mechanisms.

ii) Additionally, verification is performed on more concrete models where low level mechanisms (such as pointer arithmetic) are specified.

iii) Finally, the low level model is proved to be a correct implementation of the idealized model.

A particular class of the idealized models mentioned above are those called *security models*. These models have played an important role in the design and evaluation of high assurance security systems. Their importance was already pointed out in [58]. The paradigmatic Bell-LaPadula model [8] constituted the first big effort on providing a formal setting in which to study and reason on confidentiality properties of data in time-sharing mainframe systems.