BitML: A Calculus for Bitcoin Smart Contracts

Massimo Bartoletti
University of Cagliari
bart@unica.it

Roberto Zunino
University of Trento
roberto.zunino@unitn.it

ABSTRACT

We introduce BitML, a domain-specific language for specifying contracts that regulate transfers of bitcoins among participants, without relying on trusted intermediaries. We define a symbolic and a computational model for reasoning about BitML security. In the symbolic model, participants act according to the semantics of BitML, while in the computational model they exchange bitstrings, and read/append transactions on the Bitcoin blockchain. A compiler is provided to translate contracts into standard Bitcoin transactions. Participants can execute a contract by appending these transactions on the Bitcoin blockchain, according to their strategies. We prove the correctness of our compiler, showing that computational attacks on compiled contracts are also observable in the symbolic model.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Formal security models**; **Security protocols**;

KEYWORDS

Bitcoin; smart contracts; process calculi

1 INTRODUCTION

Cryptocurrencies like Bitcoin and Ethereum have revived the idea of *smart contracts* — agreements between untrusted parties that can be automatically enforced without a trusted intermediary [58]. These agreements regulate cryptocurrency exchanges among participants: for instance, a lottery collects bets from players, determines the winner in a fair manner, and then transfers the pot to the winner.

Disintermediation is made possible by the *blockchain*, a public, append-only record of transactions, and by the *consensus protocol* followed by the nodes to update the blockchain [27]. The execution of smart contracts relies on the blockchain to log all the participants' moves; further, the underlying logic of transactions is exploited to enable all and only the moves permitted by the contract. The consensus protocol is used to consistently update the blockchain: suitable economic incentives ensure that the nodes of the network have the same view of the blockchain. In this way, the state of each contract (and consequently, the asset of each user) is uniquely determined by the sequence of its transactions on the blockchain.

Smart contracts have different incarnations, depending on the platform on which they are based. In Ethereum, they are expressed as programs in a Turing-equivalent bytecode language. Any user can publish a contract on the blockchain. This makes the contract available to other users, who can then run it by calling its functions (concretely, by publishing suitable transactions on the blockchain). Such openness comes at the price of a wide attack surface: attackers may exploit vulnerabilities in the implementation of contracts, or may publish themselves Trojan-horses with hidden vulnerabilities, to steal or tamper with the assets controlled by contracts. Indeed, a series of vulnerabilities in Ethereum contracts [14] have caused losses in the order of hundreds of millions of USD [4, 6, 7].

Unlike Ethereum, Bitcoin does not provide a language for smart contracts: rather, in literature they are expressed as cryptographic protocols where participants send/receive/sign messages, verify signatures, and put/search transactions on the blockchain [15]. Lotteries [11, 20, 22, 47], gambling games [42], micro-payment channels [31, 48, 54], contingent payments [18, 32, 46], and more general fair multi-party computations [12, 41] witness the variety of smart contracts supported by Bitcoin.

Describing smart contracts at this level of abstraction is complex and error-prone. Indeed, establishing the correctness of a smart contract requires to prove the *computational security* of a cryptographic protocol, where — besides the usual primitives — participants can craft Bitcoin transactions and interact with the Bitcoin network. Further, these protocols often rely on advanced features of Bitcoin (e.g., transaction scripts, signature modifiers, segregated witnesses), whose actual behaviour relies on low-level implementation details. The task of proving the security of such kind of protocols requires the skills of expert cryptographers, and even in this case it is a significant effort. By contrast, working in an high-level *symbolic* model would relieve smart contract programmers from (most of) this burden, since the much higher level of abstraction would allow security proofs to be carried out with automatic tools.

Contributions. We introduce BitML (after "Bitcoin Modelling Language"), a domain-specific language for Bitcoin smart contracts. BitML is a process calculus, with primitives to stipulate contracts and to exchange currency according to the contract terms. In this respect, BitML departs from the current practice of representing Bitcoin contracts as cryptographic protocols: rather, BitML pioneers the "contracts-as-programs" paradigm for Bitcoin, by completely abstracting from Bitcoin transactions and cryptographic details. Despite the high level of abstraction, BitML can express most of the Bitcoin smart contracts proposed so far [15], e.g. escrow services, timed commitments, lotteries, gambling games, etc. The operational semantics of BitML allows for reasoning about the behaviour of these contracts in a *symbolic* setting, where the underlying cryptography and Bitcoin machinery are abstracted away.

One of our main contributions is a compiler to translate BitML contracts into standard Bitcoin transactions. Participants can perform the contract actions by publishing the corresponding transactions on the blockchain. The crucial technical challenge is to guarantee the correctness of the compiler, i.e. that the "symbolic" execution of the contract matches the "computational" one performed on Bitcoin. This correspondence must hold also in the presence of computational adversaries: otherwise, attacks at the Bitcoin level could be unobservable at the level of the symbolic semantics.

We establish the correctness of the BitML compiler through a *computational soundness* theorem [9]. More specifically, we prove that if honest participants use compiler-generated transactions,