EPiC Series in Computing
Volume 100, 2024, Pages 222–231
Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning

Scaling CheckMate for Game-Theoretic Security

Sophie Rain[1], Lea Salome Brugger[2], Anja Petković Komel[1], Laura Kovács[1], and Michael Rawson[1]

[1] TU Wien, Vienna, Austria
firstname.lastname@tuwien.ac.at
[2] ETH Zurich, Zurich, Switzerland
leasalome.brugger@inf.ethz.ch

Abstract

We present the CheckMate tool for automated verification of game-theoretic security properties, with application to blockchain protocols. CheckMate applies automated reasoning techniques to determine whether a game-theoretic protocol model is game-theoretically secure, that is, Byzantine fault tolerant and incentive compatible. We describe CheckMate's input format and its various components, modes, and output. CheckMate is evaluated on 15 benchmarks, including models of decentralized protocols, board games, and game-theoretic examples.

1 Introduction

Ensuring the security of decentralized protocols becomes even more critical in the context of decentralized finance. Once deployed on the blockchain, vulnerabilities cannot be corrected and have the potential for significant monetary loss. Various existing approaches for the analysis and verification of blockchain protocols [2, 4, 8, 14, 16, 19, 20] focus on cryptographic and algorithmic correctness or, in other words, whether it is possible to steal assets or gain secret information. However, *economic* aspects must also be considered: whether it is possible for a group of users to profit from unintended behavior within the protocol itself, leading to vulnerabilities [12]. Algorithmic game theory [7, 15] precisely captures such economic aspects.

This tool paper describes our open-source tool CheckMate[1] for the automation of game-theoretic protocol analysis. To the best of our knowledge, CheckMate is the first fully automated tool that enforces game-theoretic security. CheckMate constructs and proves game-theoretic *security properties* in the *first-order theory of real arithmetic* while ensuring that game-theoretic security is precisely captured via Byzantine fault tolerance and incentive compatibility of the analyzed protocol. As introduced in our previous work [3], Byzantine fault tolerance of a protocol guarantees that as long as users follow protocol instructions, they cannot be harmed, independently of how other users behave. Incentive compatibility ensures that the intended course of action is also the most profitable to the users, implying that no user has an economic incentive to deviate. We refer to this intended course of action as *honest behavior*, captured by an *honest history* in game theory.

[1]available at https://github.com/apre-group/checkmate/tree/lpar25

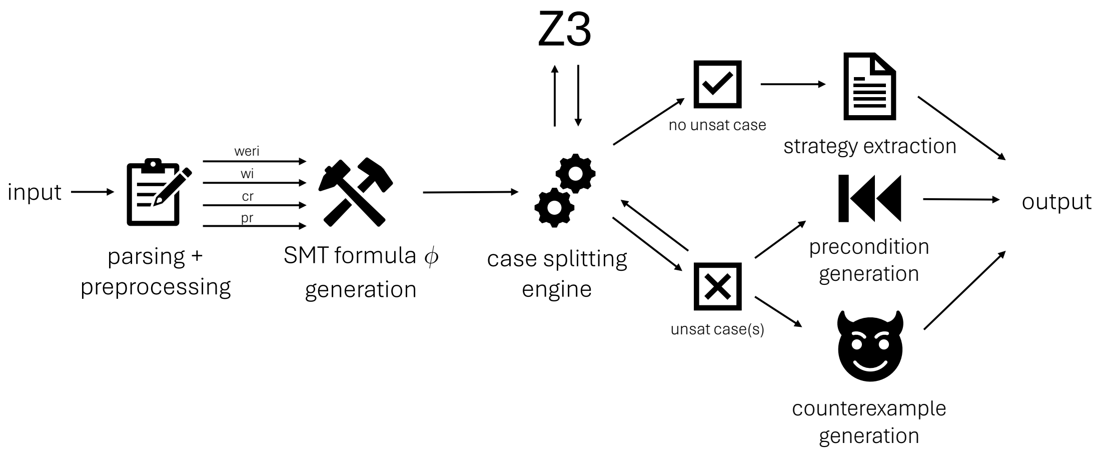

Figure 1: The CheckMate pipeline.

Following our previous work [18], inputs to CheckMate are *extensive form games* (EFGs). CheckMate translates Byzantine fault tolerance into the EFG property *weak(er) immunity*, whereas incentive compatibility is expressed in CheckMate via the EFG properties *collusion resilience* and *practicality*. As such, protocol verification in CheckMate becomes the task of proving weak(er) immunity, collusion resilience, and practicality, for which CheckMate implements novel reasoning engines in first-order arithmetic.

The purpose of this tool paper is to describe what CheckMate *can do (Section 2) and how it can be used (Section 3).* Theoretical details are covered in our previous work [3], but we also improve the algorithmic setting here. CheckMate is no longer restricted to linear input constraints, improves case splitting over arithmetic formulas, and revises counterexamples to practicality, as well as weakest precondition generation and strengthening. For efficiency reasons, CheckMate has an entirely new implementation in C++, using about 2,800 lines of code tightly integrated with the satisfiability modulo theory (SMT) solver Z3 [5]. Our experimental results show the practical gains made over our previous work and also add 7 new benchmarks to the landscape of game-theoretic security analysis. Overall, we used CheckMate to decide the security of 15 benchmarks, including five based on real-world protocols.

2 Structure and Components

CheckMate analyzes game-theoretic security of game models. Given an EFG G, CheckMate decides whether G satisfies the security properties of (i) weak(er) immunity – denoted *wi* respectively *weri* in Figure 1, (ii) collusion resilience – *cr*, and (iii) practicality – *pr*. Properties (i)-(iii) imply game-theoretic security of G [3].

Pipeline. Figure 1 summarizes the CheckMate pipeline. After parsing and preprocessing an input EFG G, CheckMate processes one honest history and security property (i)–(iii) at a time. For this history and property, CheckMate constructs *an SMT formula ϕ such that ϕ is satisfiable iff the EFG satisfies the security property* – without the need for further case analysis – with respect to the history. To this end, the SMT formula ϕ consists of three constraints, listed as (a)–(c) in the following. The constraints (a)–(c) are based on the central concept of encoding each action in the game as a Boolean variable that is set to true iff the corresponding action is taken in the game. In more detail, (a) the joint strategy constraint ensures that exactly one action is chosen at each turn. Further, (b) the honest history constraint guarantees that the chosen set of actions yields the honest history. Finally, (c) the property constraint uses a universally-quantified formula to enforce that all variables occurring in the player's pay-offs satisfying a set of preconditions also satisfy the respective security property (i)–(iii) of the EFG.

The formula ϕ is passed to the case splitting engine of CheckMate, which calls Z3 iteratively to decide whether ϕ is satisfiable. If not, case analysis is applied and the resulting constraints are added in turn to the preconditions of constraint (c) of ϕ. This iterative process is terminating as CheckMate is both sound and complete [3]. If ϕ is satisfiable, we extract a model – if required by the user – and output the result. If ϕ is unsatisfiable and no further case splits apply, CheckMate implements various actions controlled by command-line options: listing cases that violate ϕ; producing counterexamples witnessing *why* ϕ was violated; and/or computing the weakest precondition that, if added to G as an additional constraint, satisfies ϕ. We describe the main components of CheckMate using Figure 2.

Illustrative Example. The EFG in Figure 2 has two players, A and B. Nodes represent the player whose turn it is and edges their choices. On reaching a leaf, the game ends, and the pay-off *utility* for each player is given. Here, player A starts and chooses between actions l_A and r_A. If l_A is chosen, the game ends, and A receives utility $a-1$, whereas player B receives a. Otherwise, r_A is chosen, and player B continues in a further subgame. We assume $a > 0$ and specify the *honest history* of G to be (r_A, l_B): that is, we fix the "honest" choice of player A to be r_A and of B to be l_B. When analyzing whether G satisfies the security property of weak immunity (wi), CheckMate constructs the SMT formula ϕ with the following three components: (a) the joint strategy constraint given by $(l_A \vee r_A) \wedge \neg(l_A \wedge r_A) \wedge (l_B \vee r_B) \wedge \neg(l_B \wedge r_B)$; (b) the honest history constraint captured by $r_A \wedge l_B$; and (c) the property constraint of $\forall a, b.\ a > 0 \rightarrow wi(G)$.

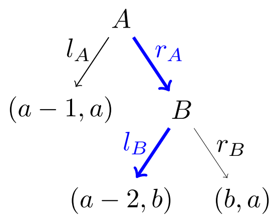

Figure 2: Game G with $a > 0$, honest history (r_A, l_B).

2.1 CheckMate Input

CheckMate takes as input a JSON file [9] with a specific structure containing the EFG to be analyzed together with its honest histories. Figure 3 shows the encoding of the EFG from Figure 2, conforming to the JSON schema of CheckMate[2]. The schema defines the structure of the input as an object with the following keys:

`players` A list of all players, represented as strings.

`actions` A list of all possible actions throughout the game, represented as strings.

`constants` Symbolic constants occurring in the players' utilities.

`infinitesimals` Symbolic constants occurring in the players' utilities that are treated as infinitely closer to 0 than the constants in `constants`. Symbolic values in utilities must be included either in `infinitesimals` or `constants`.

[2]`input.schema.json` in the repository (https://github.com/apre-group/checkmate/tree/lpar25)

`initial_constraints` Initial constraints to be enforced on the otherwise unconstrained symbolic values in utilities.

`property_constraints` Further initial constraints specifically for each security property of weak(er) immunity, collusion resilience, or practicality. This key lets the user specify the weakest possible assumptions for each security property.

`honest_histories` A list of honest histories, i.e., each history is one of the desired courses of EFG actions. Each history is the game-theoretic behavior that is (dis-)proved secure by CheckMate sequentially. An honest history is a list of actions; therefore, this key expects a list of lists of strings.

`tree` The structure of the EFG. Each node in the game tree is either a branch or a leaf. Each branch is represented by an object with the following keys:

`player` The name of the player whose turn it is.

`children` A list of branches the player can choose from. Each branch is encoded as another object with keys `action` and `child`. The `action` key provides the action that the player takes to reach `child`, another tree.

Each leaf of `tree` is encoded as an object with a single key `utility`. As leaves represent one way of finishing the game, it contains the pay-off information for each player in this scenario. `utility` contains the players' utilities, using the following keys:

`player` The name of the player.

```
{  "players" : ["A","B"],
   "actions" : ["l_A","r_A","l_B","r_B"],
   "constants" : ["a","b"],
   "infinitesimals" : [],
   "initial_constraints" : ["a > 0"],
   "property_constraints" : { "weak_immunity" : [],
                              "weaker_immunity" : [],
                              "collusion_resilience" : [],
                              "practicality" : []},
   "honest_histories" : [["r_A","l_B"]],
   "tree" : {
     "player" : "A",
     "children" : [
     { "action" : "l_A",
       "child" : { "utility" : [{ "player" : "A", "value" : "a-1"},
                                { "player" : "B", "value" : "a"  }]}},
     { "action" : "r_A",
       "child" : { "player" : "B",
                   "children" : [
                   { "action" : "l_B",
                     "child" : { "utility" : [{ "player" : "A", "value" : "a-2"},
                                              { "player" : "B", "value" : "b"  }]}},
                   { "action" : "r_B",
                     "child" : { "utility" : [{ "player" : "A", "value" : "b"},
                                              { "player" : "B", "value" : "a"}]}}]}}]}
                                                  }
```

Figure 3: CheckMate input encoding Figure 2.

`value` The player's utility. This can be any term over infinitesimals, constants, and reals provided as strings.

CheckMate Formulas. CheckMate uses infix notation in arithmetic and Boolean expressions over real numbers, constants, and infinitesimals declared in the input. It supports `+`, `-`, and `*` in arithmetic expressions with the usual meanings, but multiplication is allowed only if at least one of the multiplicands is not an infinitesimal. The Boolean expressions `=`, `!=`, `<`, `<=`, `>`, and `>=` have their usual meanings. Booleans can be combined only with disjunction spelled `|`, but this is not a limitation in practice.

Example (EFG in JSON format). In the JSON encoding of Figure 3 corresponding to the game G of Figure 2, we have the following keys. The players are `A` and `B`, the actions of G are `l_A`, `r_A`, `l_B`, and `r_B`. The only symbolic values of G are `a` and `b`. None of them are supposed to be infinitesimals; thus, they are both listed under `constants`. The only initial constraint we enforce is that `a` is strictly positive, that is, `a > 0`, as specified in the caption of Figure 2. We do not assume any property constraints, so the corresponding lists in Figure 3 are empty. As defined in Figure 2, we consider (`r_A`, `l_B`) the only honest history. In G, it is player `A`'s turn at the first internal node, which has two children. The first child, which is led to through action `l_A`, is an internal node containing utilities `a-1` for player `A` and `a` for player `B`. The other child, accessible via action `r_A`, leads to another internal node, where it is the turn of player `B`.

2.2 CheckMate Output

Given an input as detailed in Section 2.1, CheckMate analyzes each specified security property (`<current property>`) for each honest history (`<current honest history>`). To this end, CheckMate answers the following question in its output:

```
Is history <current honest history> <current property>?
```
(Q)

Figure 4 shows the CheckMate output for the input of Figure 3, when considering the security property of weak immunity.

By answering the above question (Q), CheckMate outputs intermediate logs about necessary case splits, term comparisons used during splitting, and partial results during CheckMate reasoning. Intermediate logs are displayed via indentation in the CheckMate output (see lines 4–6 in Figure 4). A partial CheckMate result indicates the satisfiability of the considered security property in the currently analyzed case (line 6 in Figure 4). Once an answer to (Q) is derived (line 8 of Figure 4), CheckMate reports – without indentation – either

- `NO, it is not <current property>`, in which scenario the EFG does not satisfy the analyzed security property and is, therefore, *not game-theoretically secure*;
- `YES, it is <current property>`, in which case the EFG with the considered honest history has the analyzed property and *may be game-theoretically secure*. If a game and a history satisfy *each security property*, that is, not only the one currently analyzed but each of the three properties of weak(er) immunity, collusion resilience, and practicality, the EFG is *game-theoretically secure*.

In addition, CheckMate can be instrumented by the user to also report on strategies (Section 2.4), counterexamples (Section 2.5), and weakest preconditions (Section 2.6) produced while answering question (Q).

```
WEAK IMMUNITY

Is history [r_A, l_B] weak immune?
        Require case split on (>= b 0.0)
        Require case split on (>= (- a 2.0) 0.0)
        Case [(>= b 0.0), (>= (- a 2.0) 0.0)] satisfies property.
        Case [(>= b 0.0), (< (- a 2.0) 0.0)] violates property.
NO, it is not weak immune.

Counterexample for [(>= b 0.0), (< (- a 2.0) 0.0)]:
        Player A can be harmed if:
        Player B takes action l_B after history [r_A]

Weakest Precondition:
        (and (>= a 2.0) (>= b 0.0))
```

Figure 4: CheckMate output for analyzing the weak immunity of the EFG of Figure 3, with counterexample and weakest precondition generation.

2.3 Case Splitting in CheckMate

The case splitting engine takes as input the generated SMT formula ϕ corresponding to the analyzed security property. CheckMate uses Z3 to determine satisfiability of ϕ. If ϕ is satisfiable, CheckMate uses the model satisfying ϕ, which is provided by Z3 and proceeds to the next reasoning engine. Otherwise, Z3 reports an unsat core, a set of constraints that are a sufficient reason why ϕ is unsatisfiable. The case splitting engine uses this unsat core to decide whether unsatisfiability is due to (i) a necessary case split on the utilities' values that has not yet been considered; or (ii) the EFG structure.

If (i), CheckMate creates two new Z3 queries: one where we add the new utility constraint to ϕ, and one with its negation. The property is satisfied only if both queries are satisfiable, which might require case splitting recursively. We record models for each case, again recursively if necessary. If (ii), CheckMate records the current case split as an unsat case together with its unsat core: these are used later for counterexample and precondition generation. If requested by the user, there is also a feature to keep exploring all cases, even after encountering unsatisfiability. This allows us to provide counterexamples to unsat cases or compute weakest preconditions to be further used in redesigning protocols without unintended behavior.

2.4 Strategy Extraction

If requested by the user and if the property was satisfied by the current honest history, CheckMate produces explicit strategies as follows. We take as input the list of cases that we divided into in Section 2.3, together with their models, and infer the corresponding game-theoretic strategy per case. These strategies provide a witness for the game and its honest history, satisfying the security property. The list of cases with their witness strategies is subsequently provided in the CheckMate output.

2.5 Counterexamples

If requested by the user (Section 3) and if the honest history violated the security property, CheckMate additionally computes counterexamples as to why the security property was violated. Depending on further flags (Section 3), one or all counterexamples for one or all unsat cases are produced. Accordingly, the counterexamples engine receives one or all unsat cases and their unsat cores. For all received cases, we study the unsat core to extract counterexamples.

To compute all counterexamples, we forbid the game choices that led to the found counterexample by adding their negation to the SMT formula ϕ's constraints and checking satisfiability. We then iterate until the extended ϕ satisfies the property. As in previous work [3], counterexamples to *practicality* are computed differently, i.e., without the use of unsat core, while following the same iterative procedure to produce all counterexamples.

Lines 10–12 of Figure 4 list a counterexample to the *weak immunity* of Figure 3. Within a counterexample to weak immunity, CheckMate reports on the harmed player p (line 11 of Figure 4) and lists the actions the other players can take to attack p. These actions cannot be prevented by the honest player p, which leads to p being harmed (line 12 of Figure 4).

In a counterexample to *collusion resilience*, CheckMate provides the group of players that profits from an attack and also lists the attack. An attack is a set of deviating actions the malicious group takes to profit, while the honest players cannot prevent these actions.

In a counterexample to *practicality*, CheckMate lists a player p, a rational subhistory r different from the honest one, and the part of the honest history after which p profits from deviating to r.

2.6 Weakest Preconditions

To extract the weakest precondition, that – if added to the set of initial constraints – makes the honest history satisfy the security property, all unsat cases have to be computed in the case splitting engine (see Section 2.3). This list of unsat cases operates as the input to the weakest preconditions routine. We apply tailored simplification steps to reduce the list of unsat cases to an equivalent and readable formula.

This only applies if the user sets the weakest precondition flag of CheckMate (see Section 3) and the analyzed honest history violated the respective security property. In this case, the computed weakest precondition is provided as output. Lines 14–15 of Figure 4 list the weakest precondition for the weak immunity of Figure 3 and history (r_A, l_B).

3 Usage

CheckMate invocations are of the form `checkmate GAME FLAGS`, where `GAME` is an input file as specified in Section 2.1 and `FLAGS` are described below. CheckMate accepts the following options to modify its behavior:

`--preconditions` If a security property is not satisfied, CheckMate computes the weakest precondition, which, if enforced additionally, would satisfy the security property.

`--counterexamples` If a security property is not satisfied, CheckMate provides a counterexample showing why the property does not hold, i.e., an attack vector. The number of considered scenarios is controlled by the `all_cases` flag.

`--all_counterexamples` If a security property is not satisfied, CheckMate provides *all* counterexamples for the violated case(s).

`--all_cases` If a security property is not satisfied, CheckMate computes *all* violated cases.

`--strategies` If a security property is satisfied, CheckMate provides evidence in the form of a strategy that satisfies it.

Additionally, the user can choose which security properties to analyze with `--weak_immunity`, `--weaker_immunity`, `--collusion_resilience`, and `--practicality`. If no property is specified, then all four of them will be analyzed by default. For instance, to generate the output shown in Figure 4, we execute

```
checkmate GAME --weak_immunity --counterexamples --preconditions,
```

where `GAME` is an input file containing the JSON encoding of the game in Figure 3.

4 Evaluation

We evaluated our tool on 15 benchmarks. Table 1 surveys our examples, with its last 7 lines listing new benchmarks compared to [3]. Out of our 15 examples, 5 describe blockchain protocols with 2, 3, or 5 players – these are the **Simplified Closing**, **Simplified Routing**, **Closing**, **3-Player Routing** and **Unlocking Routing**. The **Auction** example of Table 1 models the economic behaviors of an auction; **Tic Tac Toe** as well as **Tic Tac Toe Concise** a game of tic-tac-toe; whereas the 7 other examples of Table 1 are game-theoretic problems with 2 to 4 players. Table 1 summarizes our experimental results. CheckMate was run in default mode; that is, none of the flags described in Section 3 were set, and all four security properties were analyzed. In each of the terminating benchmarks, the current CheckMate version (version v1), presented in this paper, is significantly faster than its initial prototype (version v0) [3]. As mentioned in Section 1, this is due to our optimized and reshaped C++ implementation, as well as thanks to an improved case splitting algorithm.

Experimental Analysis. Our tool improvements make even the 5-player game **Unlocking Routing**, with 36,113 nodes, feasible to analyze. Our biggest game **Tic Tac Toe**, on which CheckMate does not terminate within one hour, is modeled in an unnecessarily huge way on purpose to show CheckMate's limitations: the majority of EFG branches could be removed for symmetry reasons. A game-theoretically equivalent pruned game tree is analyzed in benchmark **Tic Tac Toe Concise**, for which CheckMate terminates in 107.84 seconds. Scaling CheckMate further is an interesting challenge for future work.

Of the 7 new benchmarks, only **Tic Tac Toe Concise** was secure for the honest history incorporating the known tie-yielding behavior. For the game-theoretic problems G, **Centipede**, and **EBOS**, none of the security properties hold. This is not surprising as game-theoretic problems often model dilemmas, which by their nature are not secure. Surprisingly, many of the necessary preconditions were not `false`, but rather readable and short. For example, the weakest precondition to make the **EBOS** benchmark satisfy *practicality* is $2d + f \geq p$, where d, f, and p are variables occurring in the utilities of **EBOS**.

Auction violates weak immunity and collusion resilience. The model assigns the auctioneer a negative value in case the item is not being sold, while the bidders get negative values if they do not receive the item; hence, **Auction** cannot be weak immune and is reported as such by CheckMate. Ignoring the inconvenience of not selling the item, respectively not receiving it, **Auction** then becomes weak immune. This ignoring of small negative values is what weaker immunity incorporates [3]. Further, the auctioneer and one of the bidders can collude to ensure this one bidder gets the item, which contradicts collusion resilience and is also reported by CheckMate. The weakest precondition to imply security is `false`.

Finally, **Unlocking Routing** violates weak immunity for similar reasons as **Auction**, and thus also satisfies weaker immunity. It is practical, but is not collusion resilient, as it is vulnerable to the known Wormhole attack [18]. For this benchmark too, the weakest precondition to make it secure is `false`. That means the structure of the protocol has to be changed to enable security, a mere restriction of values is not enough.

Game	Nodes	Players	Histories	Time (v1)	Time (v0)
Splits$_{\mathbf{wi}}$	5	2	3	0.03	0.35
Splits$_{\mathbf{cr}}$	5	2	3	0.03	0.35
Market Entry	5	2	3	0.02	0.28
Simplified Closing	8	2	2	0.02	0.26
Simplified Routing	17	5	1	0.02	0.31
Pirate	52	4	40	1.07	27.08
Closing	221	2	2	0.34	9.60
3-Player Routing	21,688	3	1	6.83	242.54
G **(Figure 2)**	5	2	1	0.02	0.18
Centipede	19	3	1	0.07	0.48
EBOS	31	4	1	0.02	0.53
Auction	92	4	1	0.11	1.72
Unlocking Routing	36,113	5	1	10.85	478.58
Tic Tac Toe Concise	58,748	2	1	107.84	254.87
Tic Tac Toe	549,946	2	1	TO	TO

Table 1: Results of the current CheckMate (v1) versus its initial prototype (v0) from [3]. Runtimes in seconds; timeout (TO) after one hour; using a 12-core AMD Ryzen 9 7900X processor running at 4.7 GHz and 128 GB of DDR5 memory clocked at 4800 MHz.

5 Related Work and Conclusion

We describe the CheckMate tool for automating the security analysis of blockchain protocols. CheckMate complements the state of the art in protocol verification with game-theoretic security analysis, providing economic security guarantees in addition to algorithmic correctness. CheckMate differs from existing static analyzers [4, 8, 16] of Ethereum smart contracts, as these techniques merely consider cryptographic security and formally verify the Solidity [1] implementation of smart contracts. Formal methods are also used in the cryptographic verification of more general protocols [2, 10, 14], yet without game-theoretic considerations. On the other hand, existing game-based analyzers [6, 11, 13] exhibit stochastic concurrent games and provide probabilistic results about likely behaviors [11] or apply compositional techniques for simulating game behavior. Unlike [6, 11, 13], CheckMate supports SMT-based precise reasoning over symbolic utilities without predicting/simulating its EFG properties. Security analysis in CheckMate becomes a theorem-proving task in first-order real arithmetic, for which CheckMate implements novel, SMT-based techniques. With its various features and modes, CheckMate helps blockchain developers not only to analyze their protocols but also to "debug" and revise their protocol modeling and verification tasks. In particular, the counterexamples generated by CheckMate capture attack vectors to be mitigated, whereas the weakest preconditions computed by CheckMate provide constraints to be enforced in the protocols. Our experimental results demonstrate the real-world scalability of CheckMate, verifying, for example, the closing and routing phases of Bitcoin's Lightning Network [17].

Acknowledgements. We acknowledge funding from the ERC Consolidator Grant ARTIST 101002685, the TU Wien SecInt Doctoral College, the FWF SFB project SpyCoDe F8504, the WWTF ICT22-007 grant ForSmart, the Amazon Research Award 2023 QuAt, and a Netidee Fellowship 2022.

References

[1] The Solidity Authors. Solidity documentation. https://docs.soliditylang.org/en/v0.8.24/.

[2] Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif. In *FOSAD*, pages 54–87, 2013.

[3] Lea Salome Brugger, Laura Kovács, Anja Petkovic Komel, Sophie Rain, and Michael Rawson. CheckMate: Automated Game-Theoretic Security Reasoning. In *CCS*, page 1407–1421, 2023.

[4] Certora. The Certora Prover. https://docs.certora.com/en/latest/.

[5] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[6] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional Game Theory, 2016.

[7] Joseph Y. Halpern. Beyond Nash Equilibrium: Solution Concepts for the 21st Century. In *PODC*, page 1–10, 2008.

[8] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. HoRStify: Sound Security Analysis of Smart Contracts. In *CSF*, pages 245–260, 2023.

[9] ISO. The JSON data interchange syntax. ISO 21778:2017, International Organization for Standardization, Geneva, Switzerland, November 2017.

[10] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic Protocol Analysis for the Real World. In *Progress in Cryptology*, pages 151–202, 2020.

[11] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. PRISM-games 3.0: Stochastic Game Verification with Concurrency, Equilibria and Time. In *CAV*, pages 475–487, 2020.

[12] Giulio Malavolta, Pedro Moreno-Sanchez, Clara Schneidewind, Aniket Kate, and Matteo Maffei. Anonymous Multi-Hop Locks for Blockchain Scalability and Interoperability. In *NDSS*, 2019.

[13] Richard Mckelvey, Andrew McLennan, and Theodore Turocy. Gambit: Software Tools for Game Theory, 2005.

[14] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV*, pages 696–701, 2013.

[15] Martin J. Osborne and Ariel Rubinstein. *A Course in Game Theory.* The MIT Press, Cambridge, USA, 1994.

[16] Rodrigo Otoni, Matteo Marescotti, Leonardo Alt, Patrick Eugster, Antti Hyvärinen, and Natasha Sharygina. A Solicitous Approach to Smart Contract Verification. *ACM Trans. Priv. Secur.*, 26(2), mar 2023.

[17] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, 2016. https://lightning.network/lightning-network-paper.pdf.

[18] Sophie Rain, Georgia Avarikioti, Laura Kovács, and Matteo Maffei. Towards a Game-Theoretic Security Analysis of Off-Chain Protocols. In *CSF*, pages 31–46, 2023.

[19] Erkan Tairi, Pedro Moreno-Sanchez, and Clara Schneidewind. LedgerLocks: A Security Framework for Blockchain Protocols Based on Adaptor Signatures. In *CCS*, page 859–873, 2023.

[20] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *VSTTE*, pages 87–106, 2019.