Visualisation of composed cryptographic games and automatic derivation of simple cryptographic reductions

Andrew Scollin

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

The State Separated Proof (SSP) framework is used to prove the security of cryptographic protocols and constructions. SSPs use graphs as a visual medium to effectively communicate proof steps that were otherwise cumbersome to understand in traditionally written cryptographic proofs. This work focuses on developing a graph writing assistant to aid in the formulation of SSPs. We present SSP-toolkit, a web-based application specialised in making graphs for SSPs, via simple, scriptable, graph transformations.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Andrew Scollin*)

Acknowledgements

I first want to thank my supervisor, Markulf Kohlweiss, for his invaluable time and guidance during this project. Their rendition of ”(What a) Secure World” sparked my love for cryptography.

I would also like to thank Sabine Oechsner for their continuous support during this project. I would likely still be trying to make the buttons look pretty if they didn’t hold me accountable to working on the interesting parts of this project.

Many thanks to Kirthivaasan Puniamurthy and Chris Brzuska for giving me the solid foundations and the opportunity to learn from your experiences.

I also want to thank my parents, who have always supported me so much throughout my entire life.

Table of Contents

Chapter 1

Introduction

1.1 Modern Cryptography

Modern cryptography requires provable security that is mathematically rigorous. This is obvious as basing security on intuition alone is a terrible idea. However it's also not intuitive to formally prove the security of a scheme – how can we prove something is secure against an attack we haven't seen yet? This therefore seems like an impossible task, and in some sense it is (This is discussed in section 2.1.1), however *computational* security can in fact be proven under certain assumptions. If we assume that there exists a hard problem that can't be solved in polynomial time and we can show that breaking the security of a system could also solve the problem then we have some asymptotic measure of the security of our scheme. This is formalised as a *reduction* and involves an adversarial algorithm $\mathcal{A}$ that participates in a security game – this is expanded on in section 2.1.

1.2 State Separating Proofs

State Separating Proofs (SSP) [1] introduces a new technique for structuring code-based game-playing cryptographic proofs. The technique can be used to reason about constructions at different levels of abstraction by using well defined pseudocode to build monolithic *packages*. The pseudocode of a package defines the functionality of a certain component in a system and also specifies the interactions of components through function calls. This allows two or more packages to share common state variables by effectively factoring out the variables to be stored in a package that can be accessed by both. Packages and the function calls between them can be arranged as directed acyclic graphs (DAG) that represent the games used in security proofs. We will sometimes refer to the game constructions made from monolithic packages as *modular* packages as this process of monolithic package *composition* allows one to achieve modularity through the re-use of code packages. Demonstrating game equivalence is the most common way to define a security property; The negligible probability difference in an adversaries outcome between one game and another can provide a concrete way of showing if a certain scheme leaks information. Composing packages in this way not only helps to provide an understandable visualisation of the schema but also allows for clear and concise representation of equivalence between games. The specific details of SSPs are expanded upon in section 2.2.

1.3 The Problem

SSPs arose from the need to capture the complexity of key exchange protocols in a format that is not only easy for the reader to understand but easy for the cryptographer to formulate. The former goal relates to proof communication which is naturally a sought after goal in all areas of science. This project was initially motivated by the latter task of writing SSPs. Whilst SSPs enrich a cryptographers proof writing by providing a framework that serves a fluid and straightforward method for constructing proofs they also impose the challenge of illustrating the graphs. The work imposed from this task of drawing graphs is an inevitable consequence of the work saved from writing traditional proofs as the structure of the system in question has to be shown somehow. Creating graphs is albeit less work than dealing with the issues of traditional cryptographic proofs that SSPs address, but there is always room for improvement.

1.3.1 The current workflow

Diagrams.net (formerly Draw.io) is an open source diagram visualisation application and is currently the most accessible way of drawing these graphs. Whilst it has all the functionality needed to fully alter the appearance and layout of the graphs, there is a lack of functionality for the specific graph transformations that are useful in writing SSPs. Using Diagrams.net in this way can be very tedious especially since the changes that need to be applied to a graph are usually quite straightforward and could in theory be scripted. The main reason this can be frustrating is that the user has to import or create a base graph, make the changes then export the graph whilst keeping the spatial layout of the graph and the text consistent. This wouldn't be a problem if we only had a few graphs to make however with the huge number of graphs[1] needed for a proof the current workflow is just too error prone and time consuming.

1.4 Contributions

This project aims to create a user friendly solution to the failings of the current workflow for generating SSP graphs. There are many approaches that can be taken, but this project will ultimately aim to serve a web based application specifically for creating SSP graphs. The objective is for the graph editor environment to integrate smoothly into the current workflow and serve as a valuable aid to writing SSPs.

[1]This is usually the case as a consequence of *game hopping* [2]

1.5 Content

2. Background

A brief explanation of the traditional method for developing cryptographic proofs, followed by a summarised description of the state separated method. A review of the current workflow and a mention of existing SSP graph visualisation tools.

3. Initial Requirements and design

An explanation of the methods adopted in developing this tool, followed by the initial requirements analysis and an outline of the design process.

4. Final Design and Implementation

A description of the current state of the project and the functionality it provides.

5. Evaluation

A review of the usefulness of the tool, along with the revisions to be made in future iterations.

6. Conclusion

A discussion on automated proof derivation and the concluding remarks.

Chapter 2

Background

2.1 Traditional cryptographic Proofs

Since the herald of public key cryptography in the early 70's [3, 4] cryptography underwent a paradigm shift towards mathematically rigorous proofs. This was a necessity brought upon by the proposal of numerous insecure protocols that were published on a basis of intuition and were ultimately deemed insecure by-way of countering attacks [5, 6]. A cryptographer cannot assume to know every way their system will be attacked which makes proving security principles a tricky problem. The running example we present throughout this chapter is an encryption scheme. An encryption scheme is defined by three algorithms Gen, Enc and Dec as well as a specification of a key space $\mathcal{K}$, message space $\mathcal{M}$ and a resulting ciphertext space $\mathcal{C}$. The first step necessary to design any cryptosystem is defining the functionality or *correctness*. Below we provide an example definition for encryption.

Definition 1 (Symmetric encryption correctness) *a symmetric encryption scheme* $\Pi = (Gen, Enc, Dec)$ *is correct if for all valid messages* $m \in \mathcal{M}$ *and for all keys* $k \in \mathcal{K}$ *generated from Gen it follows that*

$$Dec(k, Enc(k, m)) = m$$

i.e. that decryption of the ciphertext under the same key as it was encrypted will yield the same message.

2.1.1 Designing secure schemes

The next step in designing a cryptosystem is then defining the *security* of the scheme. Making security statements requires us to define the adversaries capabilities and the set of security properties our scheme fulfils.

The Threat Model

Perfect secrecy is a notion that assumes an adversary with unbounded computational power and whilst the concept was a landmark in modern cryptography [7] in practice, perfectly secret systems are not viable as the notion imposes severe limitations on the usefulness of a perfectly secure scheme. For this reason, most cryptographic protocols are designed to be secure against *computationally bounded* adversaries that work as *probabilistic polynomial time* (PPT) algorithms. In this setup, polynomial time is established with respect to the *security parameter* (usually n or λ) of the system, which scales the difficulty of breaking a cryptosystem. This is usually implemented as a key with increasing length. We also assume the adversary is probabilistic[1] not only because some cryptography requires it (a party participating in a protocol may need to choose random keys) but also because the ability to generate randomness may provide additional power. Computational security is useful for real applications as it provides practical security; A proof to this effect conveys that it would likely take a computationally bounded adversary using state-of-the-art computers many lifetimes to break the security of a system when a sufficiently large security parameter is used.

Defining a security property

Once we have defined the capabilities of our adversary we can start building our security properties. When a cryptographic protocol between honest parties is maliciously interfered with or eavesdropped on by some adversary we can say that this adversary is conducting an *attack* on the cryptosystem.

Game-based security definitions [8] formalise the idea of security against a cryptosystem attack by setting up an experiment in which an adversary, $\mathcal{A}$, attempts to distinguish a *real* and *ideal* version of the system. The real system ("system" and "game" are interchangeable) employs the construction we want to prove secure and the ideal system exhibits perfect behaviour by design. Different cryptographic protocols have different requirements to be considered secure. We define a game G^b where $b \in \{0,1\}$ is the *security bit* (hidden from $\mathcal{A}$) that represents if the real ($b = 0$) or ideal ($b = 1$) version of the system is being presented to $\mathcal{A}$. For consistency between the following definitions and the state separated method the interaction between the adversary and a game is conveyed as $\mathcal{A} \circ G^b$. If the adversary succeeds in guessing the security bit b, the security game is won and the game G^b will output a 1, otherwise a 0 is returned. We define the notions of a negligible function and advantage as follows :

Definition 2 (Negligible function) *A function f is negligible if for every polynomial $p(.)$ there exists an N such that for all integers $n > N$ it holds that $f(n) < \frac{1}{p(n)}$*

Definition 3 (Adversarial Advantage) *The advantage of an adversary $\mathcal{A}$ that attempts to distinguish between a real game G^0 and an ideal game G^1 is defined as*

$$Adv(\mathcal{A}; G^0, G^1) := |Pr[1 \leftarrow\$ \mathcal{A} \circ G^0] - Pr[1 \leftarrow\$ \mathcal{A} \circ G^1]|$$

For example the core function of an encryption scheme is *confidentiality* – the ciphertext generated from an encryption shouldn't leak any information about the plaintext or key that generated it. To imagine this as a security definition, if we fix an adversary that, with a non negligible advantage, distinguishes the *real* probabilistic encryption of some

[1]Note that a scheme secure against a probabilistic adversary will also be secure against a deterministic adversary

IND-CPA^0		IND-CPA^1	
Initialise a shared state $k = \perp$. Then answer each adversary query as follows:		Initialise a shared state $k = \perp$. Then answer each adversary query as follows:	
SAMPLE()	ENC(m)	SAMPLE()	ENC(m)
assert $k = \perp$	**assert** $k \neq \perp$	**assert** $k = \perp$	**assert** $k \neq \perp$
$k \leftarrow\$ Gen(1^n)$	**return** $Enc_k(m)$	$k \leftarrow\$ Gen(1^n)$	**return** $Enc_k(0^{\lvert m \rvert})$

Figure 2.1: Monolithic IND-CPA games

message (using our scheme Π) from an *ideal* encryption of all 0's ($\{0\}^n$), that adversary would need to have learned some additional, potentially exploitable, information. Note that probabilistic encryption, when used correctly, means a message m submitted for encryption twice can encrypt to two different ciphertexts. This is an informal definition of Indistinguishability under Chosen-Plaintext Attack (IND-CPA), specifically Real Zero's IND-CPA. This notion suggests that confidentiality is upheld even in the case that the adversary has unlimited access to an encryption *oracle* ($ENC(.)$) that will encrypt any message under the legitimate scheme. The adversary can query the oracle as many times as they want (in polynomial time w.r.t the security parameter) before presenting a message m. A secret bit $b \leftarrow\$ \{0,1\}$ (hidden from the adversary) is chosen and the oracle returns a challenge ciphertext c that is generated from calling $ENC(m)$ on IND-CPA^b.

Killian and Rogaway [9] introduced the idea of using code to reason about games by defining their games using an informal pseudocode; These monolithic IND-CPA^0 and IND-CPA^1 definitions can be seen in Figure 2.1.

2.1.2 Proving the security of a scheme

The Code-based game-playing proofs approach by Bellare and Rogaway [10] was proposed due to a lack of consistency and conformity to mathematical rigour in cryptographic proofs. This general method not only benefits from being consistent and coherent but also makes demonstrating functional equivalence easier through *code transformations*. Showing that two pieces of code induce the same probability distribution on their common variables essentially shows that their function is identical. This is important in simplifying proofs, as the mundane steps of verifying arguments of functional equivalence can be easily verified by machine [11].

A cryptosystem simply put, is a system that employs cryptographic methods. The functions that we use to build our cryptosystem are called *cryptographic primitives*. These primitives are based on mathematical problems that are easy to formulate but are difficult to solve in polynomial time. An example of a hard mathematical problem that is considered to be computationally intractable is the discrete logarithm problem [12], which, if broken, also breaks the decisional Diffie–Hellman (DDH) assumption. The DDH assumption is used to prove the security of many several public key encryption (PKE) protocols, a principal example of which being the Diffie-Hellman key exchange

[3].

If we were granted an infinite amount of resources in storage space and computation time, there would be no such thing as security, as every key would inevitably be broken by brute force. However, in the computational security model, we reason with the hardness of problems using polynomial-time reducibility. A polynomial time *reduction* is a method of solving a problem by using the solution of another problem[2] [13]. To prove the security of a cryptosystem, we first assume there exists a PPT adversary $\mathcal{A}$ that **can** break the system. Then, if one can reduce this adversary into an efficient solution to the hard mathematical problem that the system is built on, they can show that exploiting the system is at least as hard as the mathematical problem. This is a cryptographic reduction, and it is the key step in the modern approach to provable security as outlined by Katz and Lindell [14]. This is, simply put, a mathematical proof by contradiction which is better than intuition but does still rely on the assumption that the hard problem isn't solvable in polynomial time. We also can't know *how* our hypothetical adversary $\mathcal{A}$ will break the system, therefore $\mathcal{A}$ must be used as a subroutine in some other algorithm $\mathcal{A}'$; hence this algorithm $\mathcal{A}'$ must *simulate* the execution of the security game for $\mathcal{A}$. Clearly this method is quite involved and as such it can get very complicated once we start working with modern protocols that are very complex and require multiple reductions. In our running encryption example, we examine the security of a probabilistic encryption scheme by reducing IND-CPA security to the "pseudorandomness" of the pseudorandom function (PRF) $F_k(.)$ it is built upon – a part of this process is shown in the following section.

2.2 State Separating Proofs

SSPs expand upon the code-based game-playing proof methodology and aim to provide a framework for even more concise and understandable security statements. In the SSP framework, we compose pseudocode into a series of *packages* that hold their own state variables and interact using function calls. A package P can be visualised as a node in a DAG that represents an adversary, game or reduction. Each inner edge of the graph represents the function calls between components, known simply as a packages *dependencies* denoted $[P \rightarrow]$. The "outer" interface of a package (i.e. the graphs incoming edges that have no source) are known as the *oracles* of a package and are denoted $[\rightarrow P]$. The adversarial algorithm used in a proof is also represented as a package, however it is sometimes omitted for brevity as the adversary is usually only hypothetical and therefore has no real code to reason about. The state variables of a package are private, meaning the rest of the code can only access them through oracle calls; hence the term *state separation*.

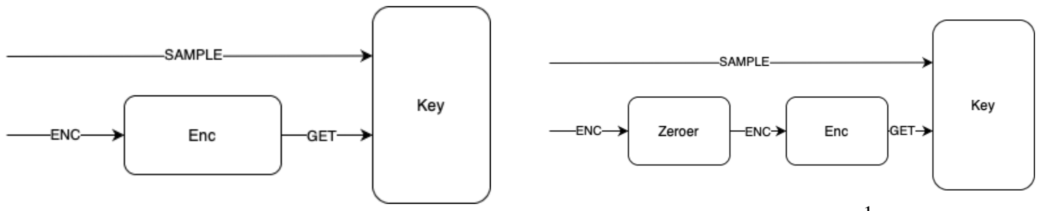
The big advantage to splitting games into packages is that we can reason about systems in a modular fashion. This makes proofs easier for the cryptographer to write and analyse as the structure of the scheme is more evident and reasoning can be inherited from a similar construction elsewhere in the paper. Another advantage to adopting this approach, is that we can give more attention to complex analysis that would normally

[2]Conversely showing the hardness of one problem by using the hardness of another problem

be hidden amongst the straightforward steps such as demonstrating code equivalences. Using this methodology we can benefit from simpler algorithm composition (interaction) through properties found in binary operations such as associativity and commutativity. It's important to note that the graphical proofs used in SSP's are not visual proofs – they are not self evident mathematical statements without extra information – rather they are used to the cryptographers benefit for their own proof communication and analysis of other state separated proofs.

2.2.1 IND-CPA

The best way to convey the usefulness of SSPs is through an example; We will continue by showing the state separated structure of the IND-CPA^b games :



(a) IND-CPA^0 (b) IND-CPA^1

Figure 2.2: The decomposed IND-CPA games

Definition 4 (IND-CPA Security) *A symmetric encryption scheme* $\Pi = (Gen, Enc, Dec)$ *is* IND-CPA *secure if for all PPT adversaries* $\mathcal{A}$

$$Adv(\mathcal{A}; \text{IND-CPA}^0, \text{IND-CPA}^1) \leq negl(n)$$

for security parameter n and negligible function $negl(.)$

Figure 2.2 shows the decomposed games and each of their packages interacting with each other. Each package is defined with specific state variables and oracles with code which can be seen in Figure 2.3. We now look at a construction for a small cryptosystem for which we want to prove that the IND-CPA property holds. Our construction uses a PRF to build a probabilistic symmetric encryption scheme. To prove IND-CPA is

```
Key
SAMPLE()
  assert $k = \perp$
  $k \leftarrow\$ \{0,1\}^n$
GET()
  assert $k \neq \perp$
  return $k$
```

```
Enc
ENC(m)
  $k \leftarrow GET()$
  $c \leftarrow\$ Enc_k(m)$
  return $c$
```

```
Zeroer
ENC(m)
  $c \leftarrow ENC(0^{|m|})$
  return $c$
```

Figure 2.3: State Separated IND-CPA syntax

upheld, we define the adversary and show them interacting with the IND-CPA games like so $\mathcal{A} \circ \text{IND-CPA}^b$. The state separation method lets us decompose IND-CPA^b into 2 packages MOD–CPA^b and PRF^b which when used together fulfil a code equivalence with our definition of IND-CPA^b. MOD–CPA^b in this case is therefore just a wrapper for the IND-CPA games that lie between the adversary and the PRF^b.

$$\mathcal{A} \circ \text{IND-CPA}^b \equiv^{\text{perf}} \mathcal{A} \circ (\text{MOD–CPA}^b \circ \text{PRF}^b)$$

then the associative property means we can complete our reduction to the security of the PRF like so :

$$\mathcal{A} \circ (\text{MOD–CPA}^b \circ \text{PRF}^b) \equiv^{\text{code}} (\mathcal{A} \circ \text{MOD–CPA}^b) \circ \text{PRF}^b$$

This process of decomposition (presented in full in the appendix of the original paper [1]) is used to show the functional equivalence of the IND-CPA^0 and IND-CPA^1 games. This means that the games are equivalent in adversarial advantage which means that IND-CPA security is upheld.

In SSPs we can also convey the information of a reduction in our graphical representation of a game. There are different ways to show this, one could colour certain nodes of the graph or draw a dotted line between the interfacing packages however the most common is by shading the region of the graph we are working to reduce. Figure 4.5 shows an equivalent formulation of the IND-CPA reduction.

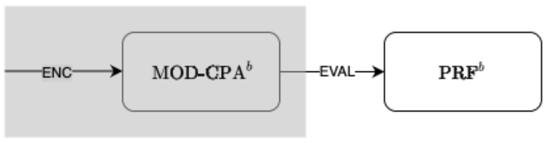


Figure 2.4: Reduction to the pseudorandomness of a PRF

2.3 Graph Theory

As this project is closely linked to the modification of graphs, there are some concepts prevalent in graph theory that need an introduction. In the most common sense of the term a *graph* is an ordered pair $G = (V, E)$ defined as:

- V a set of vertices (also called nodes)
- $E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$ a set of edges which are **unordered** pairs of vertices (i.e. an edge is associated with two distinct vertices).

In SSPs we deal exclusively with DAGs meaning the edge set E contains **ordered** tuples and never forms a closed loop anywhere across the graph. We also require that nodes and edges are *labelled*, therefore they also carry the attribute *name*.

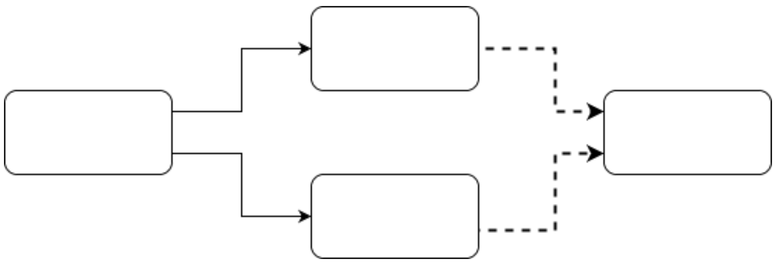


Figure 2.5: Polytree example

2.3.1 Subgraph isomorphism

A subgraph of a graph G is another graph made from a subset of the vertices and edges of G. The subgraph isomorphism problem is a task in which two graphs G and H are given, and one must determine whether G contains a subgraph that is isomorphic to H. Isomorphism in this case refers to a structure preserving bijection between two graphs meaning the vertices and edges of one graph correspond to the same vertices and edges of another. This problem has been shown NP-complete through the use of Turing machines [15]. A recursive backtracking solution to this problem was proposed back in 1971 [16] and although newer state-of-the art procedures exist that compute this matching faster [17] they use efficient typed languages that benefit from parallel code execution and are excessively complex for the performance gain in this projects use case.

2.3.2 Polytrees

A polytree is a graph with at most *one* un-directed path between any two vertices. A polytree is therefore a DAG for which there are no un-directed cycles. We deal with both polytrees and non-polytree constructions in SSPs so our tool must consider both cases. In Figure 2.5 if we are to add *either* of the dashed edges then we have a polytree, however adding both edges will make this graph non-polytree.

2.3.3 Graph rewriting

Graph rewriting (or graph transformation) is the process of creating a new graph out of an original graph algorithmically. Whilst this project need not concern itself with the state-of-the-art graph rewriting techniques, it's worth mentioning as SSP transformations[3] invariably become graph transformations when visualised. A rewriting rule (also called a *production*) is conveyed as $L \rightarrow R$ where L is the pattern to be matched within the graph and R is the graph to substitute in place of the matched pattern. An aim for this project is to make SSP transformations accessible within our tool through a scripting language, therefore it's worthwhile to mention the existence of graph rewriting systems such as GrGen [18] that are operated through the use of domain specific programming languages.

[3]Please note, our definitions of decomposition and composition of code packages aren't to be confused with definitions of decomposition and composition in graph rewriting.

2.4 Layout Algorithms

As our tool will focus mainly on the positioning of nodes and edges instead of styling options (which we leave as a task to complete in Diagrams.net) and since SSPs deal exclusively with DAGs, we put a focus on automating *hierarchical graph drawing*. A *planar* graph is a graph that optimally has no edge crossings when visualised – SSP graphs can be non-planar and planar meaning our method must deal with both; Enter, the celebrated Sugiyama layout algorithm [19]. This algorithm aims to produce a layout that spaces nodes evenly and attempts to minimise edge crossings whilst keeping edges directed in the same orientation.

The first step in the Sugiyama method involves transforming the input digraph G into a DAG and *layering* the nodes out in different horizontal levels to form a *proper hierarchy*. A proper hierarchy is made by inserting a dummy vertex at each crossing of a long-span edge with a level. We can practically skip the first sub task of this step as we are already working with DAGs, meaning we don't need to solve the *minimum feedback arc set* problem which is NP-hard [20]. Then, we are left with the problem of determining what levels to sort the nodes into. If we apply an upper bound on the number of levels we have (say one level per package at max) this is known as the *layering problem*, which is NP-complete [21].

Once we have a proper hierarchy we then perform a vertex reordering at each level to attempt to minimise edge crossings whilst keeping vertices close to their parent vertex and evenly spaced relative to the other vertices and dummy vertices. Whilst we would like to always illustrate a graph with it's optimally minimal edge crossings, the problem of reducing the crossing number of a bipartite graph to a minimum is NP-hard [22] (bipartite graph being the worst case in our application) therefore our layout algorithm must approximate this using a heuristic.

Once this order is determined we space the vertices out evenly across the level and finally redraw the graph with dummy vertices as edge points. A summary of the steps in the Sugiyama method can be seen in Figure 2.6.

2.5 Technical Background

2.5.1 MxGraph

MxGraph [23] is an open source diagramming library made by JGraph. The library we aim to use is written in JavaScript and runs natively on most modern browsers. MxGraph visualises graphs and other diagramming elements as Scalable Vector Graphics (SVG) that should be simple to work with as they have well defined geometry and coordinates. MxGraph was primarily made to serve as the framework for Diagrams.net, which is the common option for cryptographers that want to illustrate graphs for use in SSPs. This is the main reason it was chosen as the main library for displaying and altering the graphs and will be mentioned throughout this project.

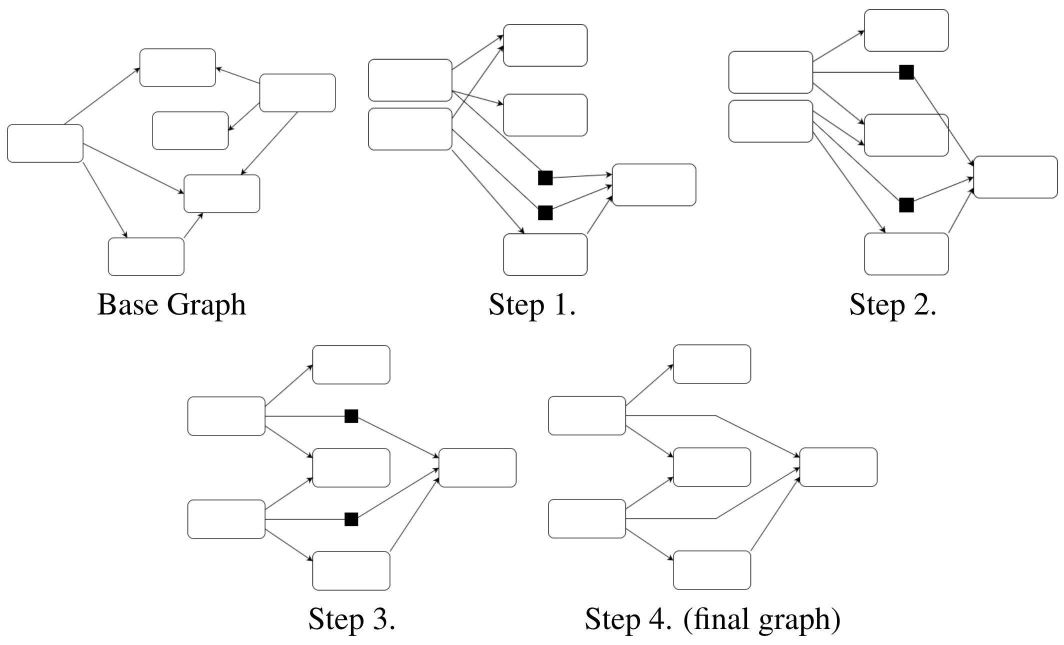


Figure 2.6: The Sugiyama method

ssp-proofviewer

The ssp-proofviewer project is a full proof visualiser for SSPs [24]. This project aimed to provide a space for SSPs to be explored without the restrictions of a traditional journal style paper and contains different examples of definitions and proofs written in the state separated style. The software developed focuses on displaying all elements of an SSP including the reasoning steps, pseudocode and graph and also provides a proof of concept for the MxGraph library. It's important to mention as the ideas and concepts from the ssp-proofviewer project are very valuable to the requirements and design of this undertaking.

2.5.2 React.js

React.js is a JavaScript library for building user interfaces (UI). The framework makes building dynamic web applications simple through the declaration of reusable UI components. This is important in reducing the size of the code base as the process of editing and displaying graphs will require translation between the verbose, XML-based *.drawio file type to a format that a user can easily edit. Handling this translated data is simple when using a framework like React as a child component accesses data through arguments passed to them meaning one parent component can handle this translation for all graphs being displayed. The other big benefit to using React is the inclusion of a virtual document object model (DOM) which, in most cases, affords an increase in performance when compared to a similarly structured UI with the same functionality written in vanilla HTML + JavaScript. This speed up will be useful when we start performing transformations on larger graphs with many edges.

Chapter 3

Initial Requirements and Design

3.1 Methodology

The approach to adopt when developing this software was an interesting decision to make as the problem is so specialised that there are only a handful of users (cryptographers that work with SSPs) that can provide valuable feedback.

Before choosing this project, there was an introductory meeting between myself, Kohlweiss and Brzuska. In the meeting they presented the state separated method and discussed the challenges they faced when making SSP graphs before introducing the ssp-proofviewer project. Reviewing this project sparked the initial idea for a web-based application which we discussed in the meeting; This impromptu brainstorming eventually turned into a weekly meeting between myself, Kohlweiss and Oechsner where we discussed what changes had been made and discussed functionality objectives for the following week. This project therefore naturally took an iterative approach throughout the design and implementation of the system.

This resembles an AGILE development process [25] as there were multiple iterations and releases. The first few iterations revolved around fulfilling the core functionality (detailed in section 3.2) to gather fast feedback in the weekly meetings whereas the last few iterations aimed to make a minimum viable product (MVP) that would actually serve as a useful tool in the current workflow.

3.1.1 Pros for using this approach

The main benefit to using this kind of iterative approach (and AGILE practices in general) is that changes in project features can be quickly implemented and evaluated. This methodology is especially fitting as having the opportunity to gain rapid feedback on new project features from two of the biggest proponents of SSPs is invaluable. This approach also means that requirements drawn from anecdotes can be made more detailed as development progresses, moreover it also makes adopting new requirements much more natural; During the initial implementation, changes had to be made to the specifications of the system which would not have been possible in a linear approach.

3.1.2 Cons for using this approach

This iterative approach did involve research, but didn't include a formal evaluation of the libraries used in development. This became an evident problem as the documentation and support for usage of the MxGraph library is severely lacking . This is a theme throughout the project and may be the biggest shortcoming of the iterative approach; At many points of development and testing there were roadblocks that came as a consequence of MxGraph being a depreciated library which then led to time estimations for completion being off. A list of the issues I faced with using this library can be found in Appendix E.

3.2 Initial requirements

"What are all the things done when writing a graphical proof?"

Whilst the working requirements changed with each iteration of the project, the initial requirements are nevertheless important to note.

3.2.1 Non-Functional Requirements

The problem at the heart of this project is that the current workflow for building and visualising SSP graphs lacks the functionality for specific semi-automated transformations therefore, at a high level, the tools most important requirement is to facilitate easy proof writing. This categorises this tool as a type of *proof assistant* which is by definition software made to assist with the writing of proofs. Völker [26] outlines several non-functional requirements for a general proof assistant which are applicable in this case. We would like to also stress the importance of some other requirements that are paramount for this project :

1. **Usability & Adaptability**
   The tool should be easy to use and accessible to all users. Since the use cases for this tool are so specialised, it's also important that the users can adapt the UI to their own preferences.
2. **Modularity & Maintainability**
   The tool should be future-proof, extensible and support existing tools and workflows (namely Diagrams.net).
3. **Help & Documentation**
   The tool should have thorough documentation and active prompts to help users understand how to use the system.

3.2.2 Functional Requirements

Below is a list of the initial functional requirements that outline the core tasks the system will help with :

1. **Editing graph templates consisting of modular packages.**
   This requirement relates to the tools ability to create and modify the base graphs

of a proof. This is just the initial structure of a game, before any transformations are applied.

2. **Decomposing a large package into a collection of smaller packages.**
   This requirement relates to the decomposition transformation commonly used in SSPs as we would often like to split up a game into multiple packages to analyse the interactions between certain components. Verifying code transformations is a sought after extension (discussed in 5.3.1), however this tool only needs to serve the user in defining *graph* transformations, meaning the incoming and outgoing edges of a package to be decomposed, are consistently mapped to the names of those in the subgraph. An example of decomposition[1] can be seen in Figure 3.1.

3. **Composing a collection of packages into another package.**
   This requirement relates to the composition transformation used in SSPs as composing multiple packages together in a single point of reference, serves to simplify reasoning with large constructions. Again however, for this project, this feature will be limited to dealing with graph transformations, meaning all selected packages and their incoming and outgoing edges are mapped to a single composed package of the users specification.

4. **Scripting transformations.**
   This requirement is a chief motive for why this project exists. Many SSP graphs tend to be quite similar within the same proof, apart from perhaps a security bit being flipped or a filter package added (A good example of this is shown in the section on *KEM-DEM* security 5.1). Therefore, a user would likely want to carry out a series of graph transformations programmatically from a collection of predefined input data.

5. **Importing and exporting graph data from/to Diagrams.net**
   This requirement is important to note as this tool doesn't aim to replace Diagrams.net completely. Outmatching the usefulness of Diagrams.net for mathematical typesetting and performing fine style editing isn't the goal of this project. Whilst, in hindsight, it definitely would be a feasible goal, since the Diagrams.net editor has open source code, the lack of informative documentation available meant it would have taken very long to implement a complete Diagrams.net clone which would've taken away from the interesting parts of this project.

These requirements are *anecdotal* and were gathered from a discussion in the first few weekly meetings between myself, Kohlweiss and Oechsner.

3.3 Initial design

3.3.1 Data Input Methods

An interesting point that Völker [26] makes is the distinction between good proof assistant design for graphical user interfaces (GUI) versus the design of text-based interfaces. Our tool deals with a data structure that is unequivocally visual; Consequently, the best

[1]This is also an example for composition, just in reverse

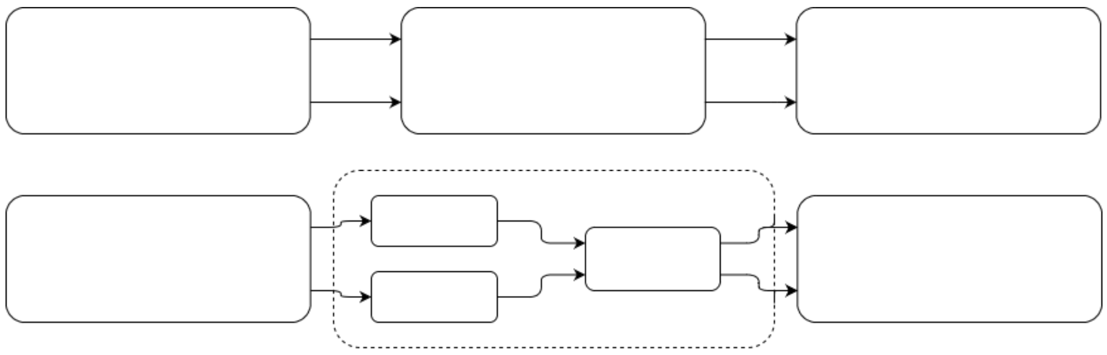


Figure 3.1: Example decomposition

data input methods in our case are accessible through the visualisation of the graphs themselves. This is easy to realise once you try and define a graph by typing the vertices and edges in a text file. Since one must define the source and target for each edge, any vertex name with multiple incoming and outgoing edges will have to be mentioned multiple times which simply isn't nice to write (or read). This means working with graphs represented in a text based format isn't as intuitive as working with already visualised graphs.

Despite this, we still must be able to script transformations which inevitably involves defining graphs in a text based way. Therefore this project will require both a text based *and* a graphical representation for the graphs a user is working on.

3.3.2 Low-Fidelity UI Prototype

Taking into consideration that the tool is focused on modifying particularly visual data structures, the design of the UI is relatively important. The first design (Figure 3.2) was sketched out using a digital stylus on Notability; This design features three prominent UI components.

The left sidebar is for creating, importing, deleting and applying transformations on the selected graph. Whilst the exact format of the transformations isn't specified, we conceptualise that this side of the screen will contain an editor for data relating to the graph that is being displayed and the transformations the user wants to run on that graph. There will also be an export option to export the transformation scripts.

The middle view port is for visualising and editing the selected graph. This graph will be rendered using the MxGraph library which should also provide the functionality for editing the graph.

The right sidebar is for managing the modular packages and their respective monolithic packages. Here a user can select what base graph they want to work on and create, delete and import new graphs to bring into the work space. There will also be an export option to export the graph data.

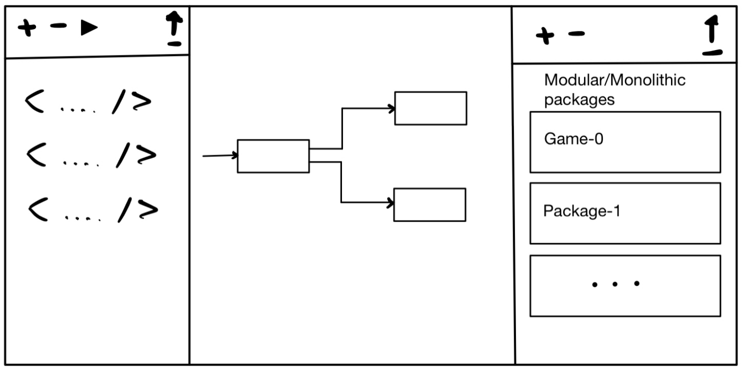


Figure 3.2: Initial UI prototype

3.3.3 Data format

The ssp-proofviewer project [24] proposed an initial data format for storing graph and proof data in JSON. This format was minimally adapted for this project to keep the possibility for future compatibility between the two environments and also because it's already in a readable and understandable format made for cryptographers[2]. There are four main aspects of the proposed data format (`monolithic_pkgs`, `modular_pkgs`, `name` and `prooftree`) however we only concern ourselves with the "`_pkg`" objects as we are, first and foremost, making a package visualisation tool. This doesn't close the door for including these objects in the project file as we won't place any restrictions on extra object keys being present.

The `monolithic_pkgs` object holds the code for the monolithic packages used in the graphs. As mentioned in our requirements (3.2.2) package code isn't involved in the logic of our transformations, however we still include package code in our data format as displaying a construction along with it's package code can be useful for the user if they want to keep all their related materials within a single project file and as it's still important to keep the possibility for future compatibility with a code equivalence checking tool. Specific monolithic package code is omitted in the format however we will work with LaTeX formatted code in the same structure as the ssp-proofviewer project.

The `modular_pkgs` object holds the structure of the graphs and their respective transformations. We differ from a generic text representation of a graph with the specification of the `oracles` object that is specifically dedicated for the graphs incoming edges. This distinction is made for readability and to conform to the same format as the proof viewer project. We will also store the scripted transformations to be run in `to_run`. The format of these transformations is specified in Section 4.8. A `history` will also be kept to afford a user the opportunity to amend an accidental execution of an incomplete chain of transformations.

Later in this project we also found the need to add a `reduction` object; This is discussed further in section 4.6.

[2]By a cryptographer!

Chapter 4

Final Design and Implementation

4.1 System Overview

As previously mentioned this is a web based application written in JavaScript using the React.js framework. The application is (currently) fully client side, as projects are saved in files that the user stores on their own machine. In its present condition, the application is a feature complete[1] prototype. This is the case as there were numerous necessary changes to the original design and whilst we ideally would like to have presented an MVP at the end of this project, it wasn't attainable given the time frame and unforeseen issues with MxGraph.

4.1.1 React Component Interaction

Following our non-functional requirement of modularity, different sections of the tool are built as React components. Figure 4.1 shows the structure of the main components of the application.

Note, however, that this diagram doesn't include the custom UI elements that were simple to build from external libraries; Examples of omitted components include the buttons, tabs and the tree view for modular packages. A full list of the external libraries used for these components is detailed in section 4.1.2. Note that to help fulfil our non-functional requirement of help and documentation, all controls are labelled with context specific icons and provide tool tips when hovered over with the mouse.

The main class that wraps all the components is the **Builder** class. This module calls a method for translating diagrams.net's native file type into project files[2] along with user controls for importing and exporting projects between their local storage and the app. This class also deals with passing graph data and keeping consensus between the components and their own stored version of the current graph being shown. Moreover,

[1]By definition, ”feature complete” suggests that the core functionality is achievable, but not yet stable enough to be released due to issues with bugs or performance.

[2]Diagrams.net's file type uses a specialised compression technique, therefore when exporting a file one must follow : File → Export as → XML – **Uncompressed**

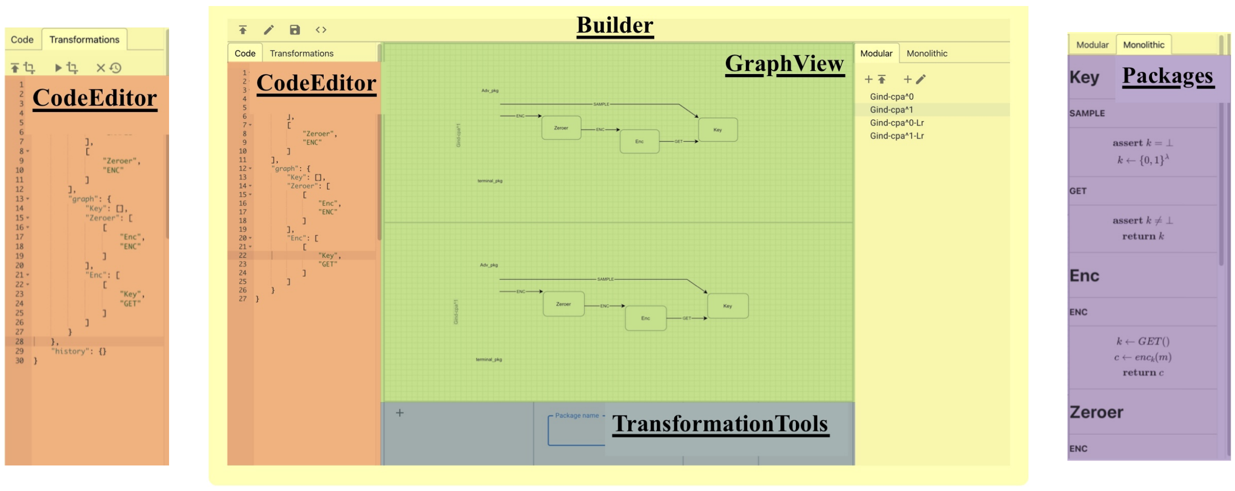


Figure 4.1: React component structure

this class implements a simple tree view for selecting a graph in the project file to be visualised and worked on.

The **Packages** class contains a space for the monolithic package code to be reviewed. Since this project doesn't validate transformations using code, this component is currently just a nicety and just aids the user in reviewing their proof steps. Down the line we imagine this section being much more fledged out and having more functionality for writing and verifying package code.

The **GraphView** class deals with interfacing between the readable JSON format and an MxGraph instance. This is achieved by providing a simple graph editing environment and view port for reviewing the result of graph transformations.

The **TransformationTools** component further aids in input data selection by providing a GUI for defining and executing transformations. Here a user can familiarise themselves with what options each transformation can be run with and they can also copy their described transformations in JSON for use in the code editor.

The specifics of the **GraphView** and **TransformationTools** classes are explained in detail in the section 4.7.2 as identifying input data for transformations is paramount to making the tool usable.

The **CodeEditor** class provides a text editor environment for editing the formatted project data discussed in section 3.3.3. The JSON specific editor was extended to feature auto-completion and custom key bindings to expedite editing graph and transformation data. A full explanation of the extra considerations into creating this specialised text editor is found in section 4.7.1.

4.1.2 Technologies used

Below is a summarised list of the most important JavaScript libraries used in this tool.

- `better-react-mathjax` was used to provide mathematical typesetting for package code.

- **`@mui/material`** provided base elements for many user interface components.
- **`react-ace`** was used to fulfil the need for a feature rich text editor.
- **`react-reflex`** was used to make re-sizeable flex boxes.
- **`react-tabs`** was used to enable tab selection.
- Finally, the **`lodash`** utility library was particularly useful in the logic of the transformations.

4.2 Layout/Styling

We implement a version of the Sugiyama layout within the tool through the use of the MxSwimlaneLayout class. This layout is applied to graphs imported from Diagrams.net and to newly transformed graphs. This was easier to accomplish than expected, however there is unfortunately lacking support for user defined layers and adding and removing swim lanes is a task that we deemed too involved for the users sake.

Despite this, our method works well and only requires small adjustments in size and positioning to be styled to the standard of a SSP. We're happy to say that every graph in this chapter (with the exception of Figure 4.4c) was styled completely within the app and only needed exporting to Diagrams.net for image conversion.

4.3 Expansion

Expansion is an additional feature conceived during the iterative process of developing this app. This feature is important to preface the transformations as it implies some restrictions on edge and package naming which consequently made the transformations more complex.

In SSPs we often have many calls to the same oracle and we sometimes also come across constructions with an arbitrary number of packages and edges. So for the sake of brevity and necessity, cryptographers will summarise a collection of d packages and edges into a single edge and package denoted with $1...d$. This is the motivation for expansion as packages summarised in this way will often need to be branched out to effectively communicate some set of proof steps.

4.3.1 Syntax

To allow for well defined expansion rules, we impose the following syntax requirement for edges and package names. Any graph imported that doesn't meet these requirements, will trigger a method in the **Builder** component that attempts to automatically resolve any illegal edges and package names.

We require that edges and packages with names that appear multiple times in the same graph are uniquely indexed by an underscore and square[3] brackets $\_[x]$, where x can be:

[3]We choose not to use curly brackets as they are used in LaTeX subscript, therefore two packages of

- A *static* number, a
- An arbitrary *static* number b, symbolised by a string containing a character in the alphabet.
- An *expandable* range $a...(a|b)$. A range of edges can only come from a static package (we call this the *base* package)
- An encoding edge with an asterisk, $a*$. Encoding edges can only exist between expandable packages.

Here we assume that an arbitrary number is just infinity. This is a lazy assumption, however, adding well-defined variables ($n \leq d \leq m$) with support for algebraic operations ($d-1, d, d+1$) would have increased complexity, especially as our range edges already made development of the core transformations much more involved, thus, we carry this assumption when referring to any arbitrary index.

4.3.2 Method

To fulfil this functionality we need a notion of *expandable chains* which are sequences of packages that **must** be expanded together given the encoded expansion information in the package and edge naming. This is a necessity as the nodes resulting from expanding a single package next to another expandable package need targets, lest encoding information is lost.

Our method identifies these chains by performing a recursive check over all nodes in the graph whilst keeping track of visited nodes so that needless checks aren't performed. Note that, since we work with polytrees, it is possible for a chain to have multiple base packages. Pseudo-code describing the algorithm for finding expandable chains is shown in Algorithm 1; This is used as a subroutine to build chains by looping over all untraversed expandable packages within the graph (See appendix B for the specific pseudocode). Extra steps for checking the graphs syntax have been omitted to keep the algorithm readable, therefore this method assumes the user has supplied a correctly named graph. Furthermore, since this subroutine is only called on expandable packages, we don't need to restrict the user in using the edge range naming convention between two static packages, as these edges won't be checked by the algorithm.

We allow a user to expand a series of packages out to any integer value less than the least expandable package in the chain; The graph shown in Figure 4.2 (a) has a chain with an arbitrarily expandable package, however, is it restricted to only expanding by a value of 3 by the other expandable package. Here we see the function of the different edge naming, An *encoding* edge is used for "dynamic" edge mapping – as opposed to static edges that are duplicated but always map to the same package once expanded. The presence of an encoding edge is only valid when its source and targets are expandable packages; The base index of the source node, the index of the encoding edge name and the base index of the target node will iterate in parallel whilst being expanded. An example of expansion with multiple different encoding edges can be seen in Figure 4.3. This allows a user to script an expansion of packages with specific edge sequences;

the same name with different subscript are considered to have different names.

Algorithm 1 `findChain`, **Input:** $v, visited, isBase$

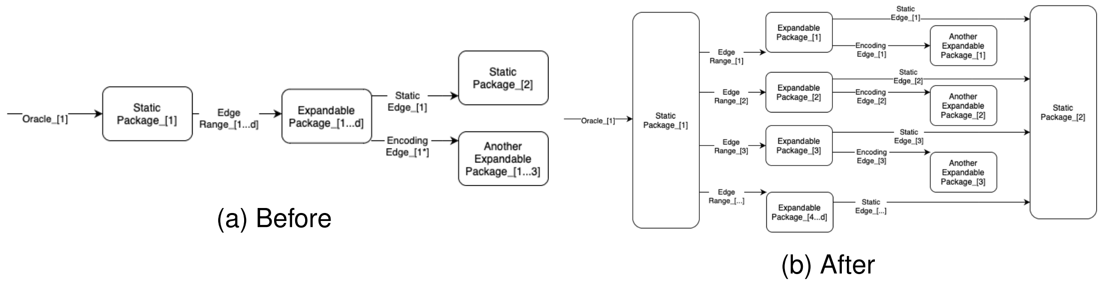
```
1: $chain = \{\}$
2: if $isBase == \top$ then
3:     return $chain$
4: end if
5: $visited.add(v.name)$
6: for $e \in v.outgoing$ do
7:     if $e.target \notin visited$ and $'...' \in e.target.name$ then
8:         $chain.add(findChain(v, visited, \bot))$
9:     end if
10: end for
11: for $e \in v.incoming$ do
12:     if $e.source \notin visited$ then
13:         if $'...' \in e.source.name$ then
14:             $chain.add(findChain(v, visited, \bot))$
15:         else if $'...' \in e.name$ then
16:             $chain.add(findChain(v, visited, \top))$
17:         end if
18:     end if
19: end for
20: $chain.add(v.name)$
21: return $chain$
```

Later we will look at a real example of this with the proof of Yao's garbled circuit (See section 5.2).

When expanding packages using encoding edges there is sometimes ambiguity between the newly made packages and the rest of the package range that isn't visualised. In our dynamic example, ”Another Expandable Package_[5]” would be incomplete as we haven't got an ”Expandable Package_[5]” to draw an ”Encoding_1” edge from, therefore we include the option to add ”ghost” edges and packages to convey how long the sequence continues for (in this case, to an arbitrary length). These ghost edges can be seen throughout SSPs, however, they are usually styled as a vertical sequence of dots. Adding these dots increases complexity when it comes to performing our graph layout, therefore, we've excluded this feature for now, and leave it as a task for the user to change out the placeholder packages when performing their final graph styling.

Similarly to the graph transformations, expansion is scriptable. The input data required for an expansion is a single expandable package that specifies the chain to be expanded, the value of the expansion and a Boolean to specify if ghost edges should be added or not.

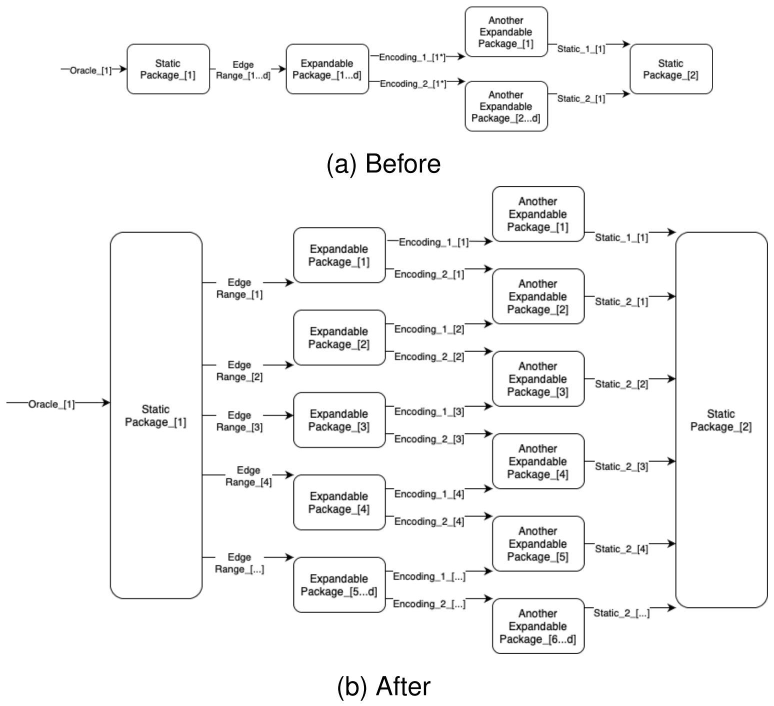
We will continue with a full analysis of the transformations implemented in this project as an understanding of their function is necessary to fully appreciate the design and implementation of **TransformationTools**.



(a) Before

(b) After

Figure 4.2: Simple expansion with a value of 3



(a) Before

(b) After

Figure 4.3: Dynamic expansion with a value of 4

4.4 Decomposition & Composition

Decomposition and Composition are relatively simple graph transformations, however, they are likely going to be the two most commonly used transformations, therefore they still need special consideration.

The edge naming rules imposed by expansion make these transformations easy in some ways and difficult in others; We benefit from the fact that edges are uniquely named but suffer from the imposed complexity of edge ranges.

4.4.1 Decomposition method

Decomposition involves swapping a target package in a graph with another graph that has a matching oracle interface. This is simple, in theory, as we assume packages are uniquely named since we require them to be indexed. Therefore, we just need to match the packages of the incoming oracles to their relative packages in the graph to be fitted, however, edge ranges impose the challenge that one or more edges in an edge range may be extracted out of the range in the graph we are fitting. An example of this specific case can be seen in Figure 4.4. Therefore when we resolve a decomposition we follow these steps :

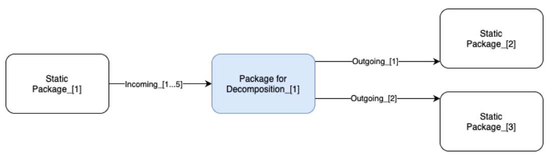
1. Build separate collections for the target packages incoming and outgoing, non-expandable and range edges.
2. Build the same list for the graph that is to be fitted; Here we are adding edges as ”expandable” if their base names exist in the base names of target expandable edges. This list is built from the fitting graphs oracles and the edges with no destination (we automatically destine these edges for a placeholder ”terminal_pkg” in the app).
3. We resolve and check that each non-expandable edge is mapped from a package in the graph to a package in the fitting graph at least once; We allow an edge to be mapped more than once but throw a warning for the user.
4. We then do the same for each expandable edge. We first check that the edge is mapped in at least one target range (recall that the limit of an ”arbitrary” range is just infinity) then we resolve them based on which range fits that edge; Again, we allow an edge to resolve to multiple ranges, but we will warn the user when this happens.

There is a big qualm with this process however; Since we assume that arbitrary numbers are just infinity we cannot accurately fit an arbitrary index into an arbitrary range. For example, an edge indexed with $_[d+1]$ will map to the range $_[1...d]$. This is an unfortunate consequence of making this assumption, alas if we didn't overlook this, the interesting parts of this project wouldn't have been possible given the time constraints, thus we leave it for future work.

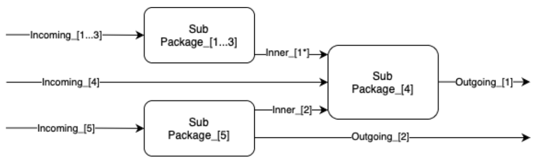
Decomposition is scriptable and takes a target package name and the graph to be fit inside the target package.

4.4.2 Composition method

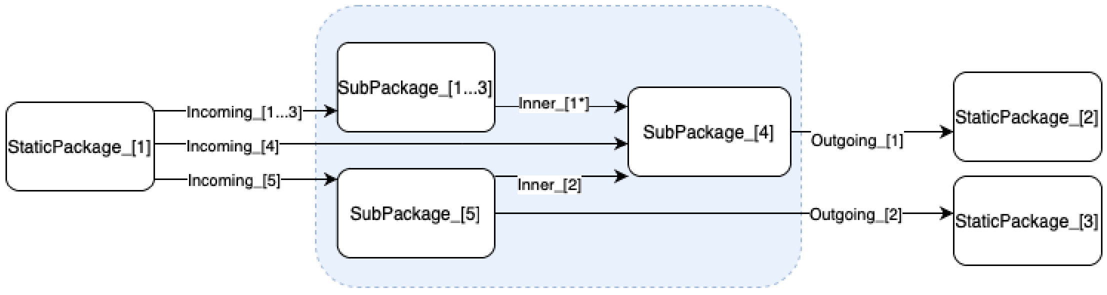
Composition on the other hand is much simpler as our input data is just a list of package names to be composed and a name for the newly composed package. All incoming and outgoing edges of the selected packages are just redirected to a single package and edges between any selected packages are removed. If edge names overlap and the indexes of the edges are incremental the transformation will absorb the indexes into an edge range. For example, a composition on Figure 4.4 (c) of the packages SubPackage_[1...3], SubPackage_[4] and SubPackage_[5] can take us back to the graph in Figure 4.4 (a).



(a) The graph (the target package is highlighted in green



(b) The graph to be fitted



(c) The graph with decomposed package

Figure 4.4: Decomposition with edge ranges example

4.5 Subgraph Substitution

The "Substitute" transformation offers the ability to perform a general *subgraph substitution* (or, in cryptography sense, define and evaluate the outcome of a code equivalence[4]). This transformation basically allows one to use a graph rewriting rule of the form $L \to R$. Note that we require that L and R are (respectively) *connected* graphs since there is no way to automatically generate the users intended graph if either L or R is disconnected. One may think that the implemented subgraph substitution makes decomposition and composition useless, however the restriction of having connected graphs for L and R makes general decomposition and composition impossible. The restriction of having connected L and R is discussed more in the concluding remarks on automated proof derivation (See section 6.1).

This task involves solving the aforementioned subgraph isomorphism problem, which we solve through an altered implementation of the algorithm for pattern matching first proposed by Ullmann in 1976 [16].

To effectively describe our algorithm we introduce a non-conventional attribute of vertices and edges called *basename* which is just a non-indexed *name* and the *index* attribute, which is self explanatory. These attributes are mentioned as our pattern L and the substitution R, are just non-indexed subgraphs therefore we match nodes in the graph over their base names. The main method used for checking the "completeness" and for resolving the external edges of a potential pattern is described in Algorithm 2. This algorithm is used as a subroutine within the substitution transformation that runs `checkComplete` over all non-traversed, candidate vertices v in the supplied graph.

This parent process will revert any edge changes if the subroutine returns $\perp$ at any time. It also resolves inner edges and keeps track of all package names within all the

[4]Again, we don't actually deal with code in our implementation, thus we use the term "subgraph substitution"

matched patterns and will automatically resolve instances of these in other matches after every node is checked; This covers the case where two patterns share an outgoing and incoming edge.

A quirk with this method is that a subgraph will match if it *only* contains external edges and vertices that appear in *L*, however there is no check that *every* node and edge of *L* is matched, therefore the parent function also performs an extra check to ensure that every edge and vertex was covered in `checkComplete`. This is the default behaviour, however we let the user decide if this check is performed via a ”partial” matching flag as there may be some ambiguity in graph structure from ghost packages generated by a preceding expansion. Therefore the input data for scripting a substitution is *L*, *R* and the optional Boolean for partial matching.

Algorithm 2 `checkComplete`, **Input:** $v, L, R, visited$

```
1: $visited.add(v)$
2: for $e \in v.incoming$ do
3:     if $e.basename \in L.V[v.basename].incoming$ then
4:         $e.target = R.E[e.basename].target + v.index$
5:     else if $e.source.basename \in L.V$ and $e.source \notin visited$ then
6:         if no edge in $L.V[e.source.basename].outgoing$ matches $e$ then
7:             return $\perp$
8:         end if
9:         $complete = checkComplete(e.source, L, R, visited)$
10:        if not $complete$ then
11:            return $\perp$
12:        end if
13:    else if $e.source \notin visited$ then
14:        return $\perp$
15:    end if
16: end for
17: for $e \in v.outgoing$ do
18:    if $e.basename \in L.V[v.basename].outgoing$ then
19:        $e.source = R.E[e.basename].source + v.index$
20:    else if $e.target.basename \in L.V$ and $e.target \notin visited$ then
21:        if no edge in $L.V[e.target.basename].incoming$ matches $e$ then
22:            return $\perp$
23:        end if
24:        $complete = checkComplete(e.target, L, R, visited)$
25:        if not $complete$ then
26:            return $\perp$
27:        end if
28:    else if $e.target \notin visited$ then
29:        return $\perp$
30:    end if
31: end for
32: return $\top$
```

4.6 Defining Reductions

The ”Reduce” transformation aims to let a user define a reduction by automatically shading a selection of packages. We preface this section by admitting that the ideal highlighting transformation we outline was only designed and was not fully implemented. Only a proof of concept was achievable due to issues with MxGraph components and the coordinate system they use; These issues are discussed in Appendix E.

When defining a reduction there are two tasks at hand: the first is to highlight the packages to be included within the reduction, the second is to change the bit (superscript) of the package being reduced. We achieve the latter goal by allowing the user to specify a string for the new bit, then performing the bit swapping using Regex.

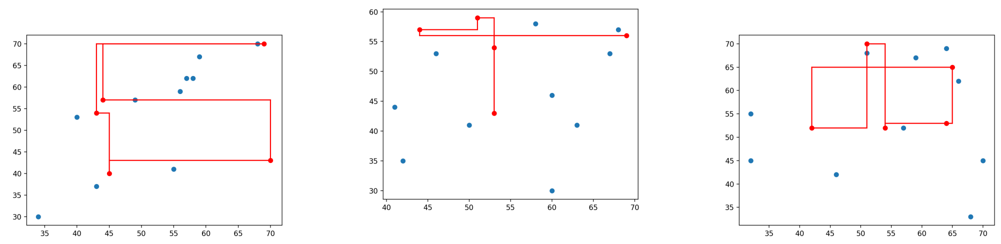
When using highlighting – as shown in Figure 4.5 – the former task is similar to finding an axis-aligned bounding box for all the specified packages and their inner edges. The task differs slightly as we don’t just want a bounding box that is just the maximum and minimum x and y coordinates; Rather we want every point to be traversed using an optimally minimal number of 90° turns. We define *target points* as package coordinates from the furthermost corner from the centre of the whole graph and as the absolute coordinates of edge entry, exit and bend points.

Initially we tried to loop over every target point; Then from the current point $(x1,y1)$, either walk to the x coordinate $(x2,y1)$ or walk to the y coordinate $(x1,y2)$ of the next point; Then walk to the next point $(x2,y2)$ and repeat. We choose to walk to either x or y depending on which engulfs less points that aren’t targets, we can do this by comparing the number of non-target points that intersect with the triangle made from the current target point, either of the middle points and the next target point. This would supply us with a polygon that, when filled in with a simple semi-transparent highlight, would match our reduction shape very well. Once the highlighting has been added, we could then check for the intersection of any packages or edges that don’t belong to the reduction and highlight them in opaque white.

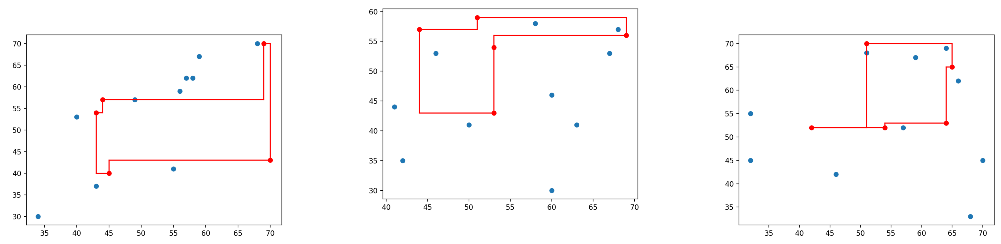
This is the most simple way of making an automatic reduction, however in the early stages of testing, we thought this wasn’t feasible to implement as the MxShape prototype threw an error when we tried to stencil a new custom shape. We naïvely assumed that this was because multiple lines were being crossed whilst stencilling the path; however we later found that MxShape didn’t work regardless of whether lines were crossed or not. Before finding this out, we tried the following adaptation to our approach to compute our bounding boxes without crossing lines :

1. Perform the Graham’s Scan algorithm on the selected target points to determine the convex hull that encapsulates all the packages and edges to be ”reduced”.
2. Then carry out the same steps as above on the hull, checking for the *presence* of any target point that may be missed instead of the minimum number of non-target points.

We can be sure that step 2 of our adapted procedure produces a bounding box as we search over the convex hull of our target points, so if any point isn’t included when we direct the path in one axis towards the next point, it definitely will be contained if we



(a) Original traversal method



(b) Adapted traversal method

Figure 4.5: Three different randomly chosen sets of target points (in red) and non-target points (in blue)

direct the path along the other axis.

Figure 4.5 shows a simple proof of concept for both of our approaches, implemented in Python. It's clear to see that both methods would fit the criteria for a reduction shape however the original method is consistently better at only capturing target points.

Unfortunately however, the drawbacks that come with using MxGraph restrict us to the option to just colour packages which isn't ideal as we prefer to use a bounding box highlight, nevertheless this was in fact successfully implemented.

There is a bit of ambiguity in the "reduce" title given to this transformation as there exists the case when a user, working with game G^0, wants to perform a reduction to achieve G^1 but they need an intermediary graph G^b so they can effectively convey an assumption inherited from the reduction. For example the real and ideal versions of G may require a change in inner function calls, which currently isn't possible in a single step. Therefore, the ambiguity here is that, to perform a single reduction, a user must first perform a "reduce" transformation over the necessary packages and a bit b to produce G^b, then they must perform a substitution to match their reductions assumption, then they must perform another "reduction" over an empty list of packages and the bit 1 to generate the two graphs they seek.

This transformation is scriptable, therefore the user must supply a list of package names to be included within the reduction highlighting and a string detailing the superscript they wish to change the reduced package to. Note that the `reduction` object within the graph data specifies what packages are included in the reduction. This currently will just change the colour of any packages within it to a light blue.

4.7 Data Selection UI

4.7.1 Text based UI

The **CodeEditor** class serves as a text editing environment. There are two different instances of this editor implemented in the application, one for modifying graph data and one for modifying transformation data. Both environments are fully featured, high performance, code editors that benefit from live JSON linting.

In many cases, different components within a cryptosystem will make multiple calls to the same oracle and we will also sometimes see multiple instances of similarly named packages. Whilst renaming a *single* package or edge is easier in the visual representation of a graph, it isn't as easy to rename *multiple* edges and packages. Therefore, the find and replace all shortcut present in the **CodeEditor** is especially useful.

The **CodeEditor** instance that is used for modifying graph data features dynamic auto-completions that let the user quickly reference the monolithic packages and edge names that are mentioned anywhere in the project. The **CodeEditor** instance dedicated to scripting transformations uses auto-completions to aid in structuring the different transformations and for referencing the names of modular packages.

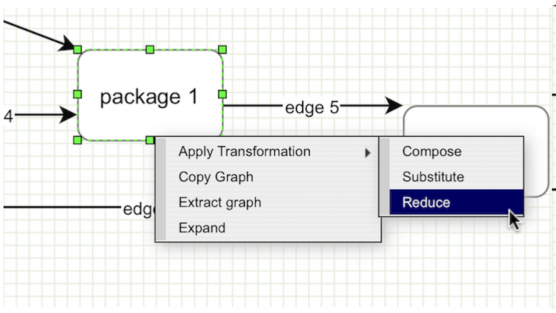
4.7.2 Graphical UI

The **GraphView** component is the area where graphs are visualised and worked on. Whilst the **CodeEditor** component is powerful in its own right, for the aforementioned reasoning on GUIs versus Text based UIs (3.3.1) the **GraphView** will undoubtedly be the main section a user will be working with to define graphs. Hence, a user can:

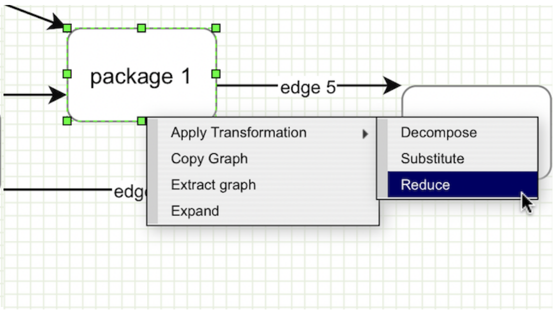
- Add packages by dragging and dropping the rectangle in the toolbar.
- Define edges by dragging from the centre of a package.
- Rename specific packages and edges by double clicking on them.
- Select multiple packages by holding the command key.
- Delete selected packages by pressing the backspace key.
- Copy and paste between different selected graphs using the generic key bindings.
- Undo and redo graph modifications using the generic key bindings.

Although these features seem trivial to implement, using the MxGraph library without being able to access the MxEditor class meant this process was quite painstakingly involved (See Appendix E).

A right-click context menu (Figure 4.6) was added to provide easy access to the features available in **TransformationTools**. Selecting ”Expand” or any of the options within the ”Apply Transformation” sub menu will trigger **TransformationTools** with appropriate control options. The context menu is selection specific, so when user selects multiple nodes they will be given the option to perform a composition, but when a user selects one node, they will be given the option to decompose.



(a) Context menu on multiple selection



(b) Context menu on single selection

Figure 4.6

The ”Copy Graph” option takes the users selected nodes and edges and translates the subgraph they define into the JSON data format for use in the scripting of transformations. A subgraph selected this way can also be extracted using ”Extract Graph” which adds the selected model to the modular package list so one can pre-define and tweak all the necessary subgraphs for use in their scripted transformations. These two options are integral to the envisioned workflow this tool is aiming to serve. In SSPs we want to show the equivalence of two games, and games that are practically identical will undoubtedly share some common structure, thus, being able to easily select parts of one game that can be saved, edited and utilised *anywhere* in the tool is invaluable.

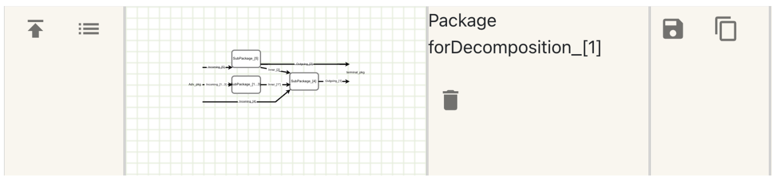
The **TransformationTools** component expands when a transformation is triggered and will always have options for saving the transformed graph and for copying the JSON for the transformation script in the bottom right panel. A different set of controls will appear depending on the type of triggered transformation.

The controls for **decomposition** (Figure 4.7a) include the option to select a subgraph either from importing a .drawio file or a JSON file matching our data format. We also provide the option to select a subgraph from any graph that has already been defined within the project file. Once chosen, the subgraph also appears in a small viewport within transformation tools. The user can also change the selected package to be decomposed.

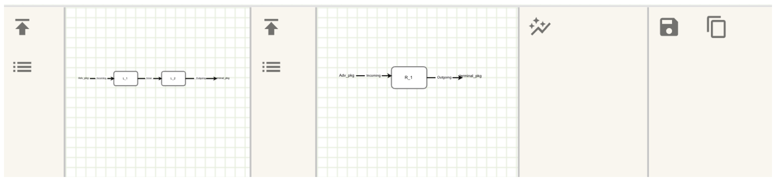
The controls for **substitution** (Figure 4.7b) are similar to those of decomposition; there are options for importing and selecting both the L and R graphs. Each selected graph has its own viewport and there also exists a button for specifying partial matches.

The controls for **composition** (Figure 4.7c) include the option to specify the composed package name using a text box input as well as options to add or remove packages to be composed. These controls are identical to those for specifying a **reduction**.

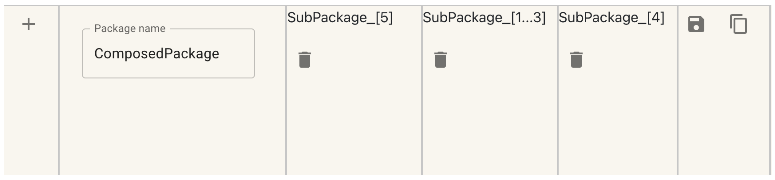
Whilst it’s not itself technically a transformation, **TransformationTools** also has controls for **expansion** (Figure 4.7d). As previously mentioned a part of expansion involves finding the expandable chains of packages and edges, therefore the controls will show a different value slider for each of the expandable chains in the selected graph to let the user visualise live how the graph expands and collapses.



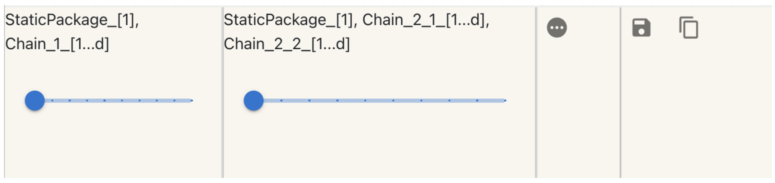
(a) Decomposition



(b) Substitution



(c) Composition


StaticPackage_[1], Chain_1_[1...d]
StaticPackage_[1], Chain_2_1_[1...d], Chain_2_2_[1...d]

(d) Expansion

Figure 4.7: Different controls for **TransformationTools**

4.8 Scripting transformations

The ”Transformations” tab in the **Builder** component holds **CodeEditor** space for writing and running transformations.

The format of the different types of scripted transformations is shown in Appendix A.2. Here, the keys of the `to_run` object indicate the name for the new graph generated by running the linked transformation. It’s possible to have a `to_run` object inside a `to_run` object; This allows the user to flexibly script multiple transformations on either a single base graph or on the graph generated from a previous transformation.

A sequence of transformations can be run via the controls atop the editor environment. If at any point a transformation fails, the process will only generate graphs and a history up until the illegal transformation which makes amending mistakes relatively straightforward. These controls also allow a user to easily import new transformations or clear their transformation history.

Chapter 5

Evaluation

We will now provide example workflows for constructing SSPs for the security of some real systems to illustrate the usefulness of the tool.

5.1 KEM-DEM

5.1.1 Background

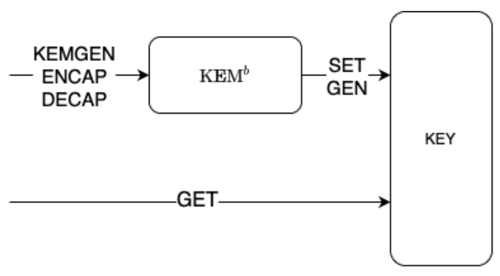
The *KEM-DEM* scheme is a communication technique that unites public key cryptography with the efficiency and large message space of private key encryption. This construction provides a means for *hybrid* encryption in which a key encapsulation mechanism (*KEM*) first is used to fix a random *session key* that is then utilised in an efficient data encapsulation mechanism (*DEM*) to encrypt the actual message m. In general, a ciphertext generated for use under $KEM||DEM$ is defined as follows:

$$c := \varepsilon_{pk}^{\text{asym}}(K)||\varepsilon_{K}^{\text{sym}}(m)$$

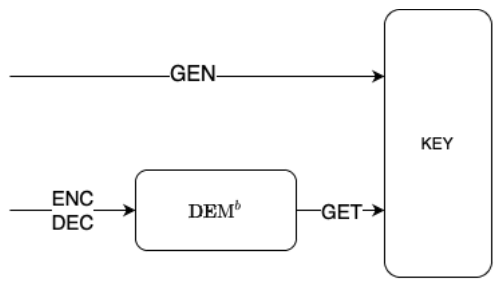
The *KEM* component of this construction facilitates a *key agreement protocol* which is realised via a PKE scheme. This is necessary as we require that the parties who wish to communicate using a session key can confidentially agree on such a key over a public channel.

The *DEM* component of this construction is realised by a private key encryption scheme. This is because true PKE schemes have a few construction-inherent limitations; Firstly, it is very slow to encrypt a long message as PKE operations require calculating exponents that are on the order of 10's to 100,000's times slower than the calculations used in symmetric encryption schemes. Secondly, using a scheme solely based on PKE primitives fails to provide *forward secrecy*; This feature requires that a scheme protects past communication in the event that a long-term secret is compromised.

Using ephemeral keys for use in a DEM solves both of these problems as long as the KEM and DEM components are both secure in their own right and are implemented correctly within the hybrid construction. There are many notions of security that relate



(a) The KEM game



(b) The DEM game

Figure 5.1

to the KEM-DEM construction [27] however we will be, briefly, outlining the proof for indistinguishably under chosen ciphertext attack (IND-CCA) security as presented in the canonical SSP paper [1].

We decompose the KEM and DEM security games into two packages that both interact with a KEY package that stores the shared key. This KEY package is identical to the one described in section 2.2.1 apart from the extra feature of being able to have its key explicitly set by calling $SET()$; This is required by construction, from the operation of the KEM. As with our example with IND-CPA we want to show an equivalence between the real and ideal KEM-DEM game. We do this by interfacing the adversary $\mathcal{A}$ and the system with a wrapper MOD-CCA that calls the relevant games. The KEM^b package consists of three polynomial-time algorithms, which call on a secure KEM, η. The DEM^b package consists of two polynomial-time algorithms, which call on a secure DEM, θ. Both composed games and the extension on the KEY package are shown in Figure 5.1. MOD-CCA provides an interface for the KEM-DEM system and as such provides oracles that are identical to those of a PKE CCA game[1], namely $PKGEN, PKENC(m), PKDEC(c')$. See Appendix C.1 for all of the relevant packages code.

5.1.2 Required Graphs

As we are using a hybrid scheme to show the equivalence of the whole construction we must make three reductions as we assume the state of the base graph then show it's equivalence to the graphs when $b = 0$ and $b = 1$ for both KEM and DEM; Therefore we first idealise KEM, then we idealise DEM, then we de-idealise KEM.

This requires us to produce seven graphs (Figure 5.2) which we can achieve by using six reduction transformations to capture the graphs required for the three reductions and and two substitution transformations to match the correct oracle calls for the ideal and real KEM games.

The script and graphs used to run this sequence of transformations is included in Appendix C.2. Note that the only styling applied in Diagrams.net involved removing placeholder packages and adding mathematical typesetting.

[1]The function of MOD-CCA and PKE game is shown perfectly indistinguishable in Appendix D of [1]

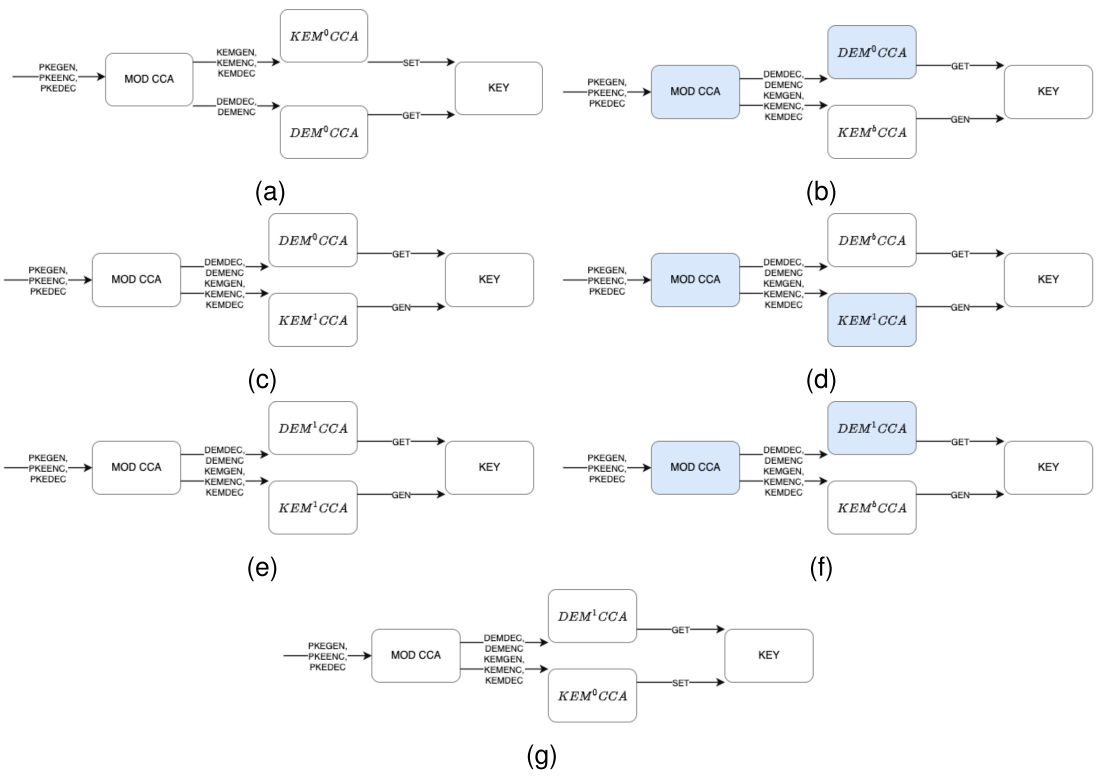


Figure 5.2: The *KEM-DEM* proof

One problem we identified is apparent in 5.2b – our tool swaps the *KEM* and *DEM* placement due to the substitution transformation we apply at this stage. We aim to remedy this in future iterations of the tool however this is a complex task as this involves changing the tools automatic layouting which is primitively bound to the MxGraph library.

5.2 Yao's garbled circuit

5.2.1 Background

Garbling schemes represent a function F by a randomised function $\hat{F}$ such that $\hat{F}(x)$ reveals $F(x)$ and no additional information about x. Such a scheme allows two mistrusting parties that want to securely compute a function over their private inputs without the presence of a trusted third party thereby enabling Secure multi-party computation (MPC).

Multiple applications for garbling schemes exist; Yao's millionaire's problem [28] was the first presented problem for 2PC. In this problem two millionaire's want to see who is richer without revealing either of their net worth – essentially computing $F(x,y) = x \geq y$, without revealing x or y. The solution initially proposed by Yao is impractical as it is exponential in time and space, however in 1986 Yao presented the first construction of a *garbled circuit* [29] for practical 2PC.

The details of the construction of a garbled circuit are quite lengthy, therefore we will only describe the necessary definitions of correctness and the structure of the semantic security games involved to give sufficient background. The game definitions are taken from a recent SSP for Yao's Garbled Circuit presented by Brzuska and Oechsner [30] – we encourage readers to review their paper for a complete explanation into the proof for garbled circuits. The (simplified) operation of such a scheme is as follows; Here we use Alice and Bob as the two mistrusting parties:

1. Represent the function f to be computed as Boolean circuit C.
2. Alice garbles the circuit C into $\tilde{C}$ (gb) and encodes their input x_1 into $\tilde{x}_1$ (en) and sends $\tilde{C}$ and $\tilde{x}_1$ to Bob.
3. Alice and Bob run a protocol to encode Bob's input x_2.
4. Bob evaluates the circuit $\tilde{C}$ on $\tilde{x}_1||\tilde{x}_2$ and decodes the output $\tilde{y}$ to y, then sends y to Alice.

Definition 5 (Garbling scheme [31]) *A circuit garbling scheme consists of 5 probabilistic, polynomial-time algorithms:* $\mathbf{gs} = (gb, en, de, ev, gev)$*, for circuit garbling (gb), input encoding (en) and output decoding* (de)*, circuit evaluation (ev) and garbled circuit evaluation (gev).*

Definition 6 (Garbling scheme correctness [31]) *Let* $k \in \mathrm{N}$*. A garbling scheme* $\mathbf{gs} = (gb, en, de, ev, gev)$ *is correct if for all circuits C and inputs x,*

$$Pr_{(\tilde{C},e,d)\leftarrow\$gb(1^k,C)}[ev(C,x) = de(d, gev(\tilde{C}, en(e,x)))] = 1$$

Definition 7 (Garbling scheme security [30]) *Let d be a polynomial in n. Let* $\mathbf{gs} = (\boldsymbol{GB}, \boldsymbol{GEV})$ *be a garbling scheme.* **gs** *is secure if there exists a PPT simulator* ***SIM*** *such that for all PPT adversaries* $\mathcal{A}$*,*

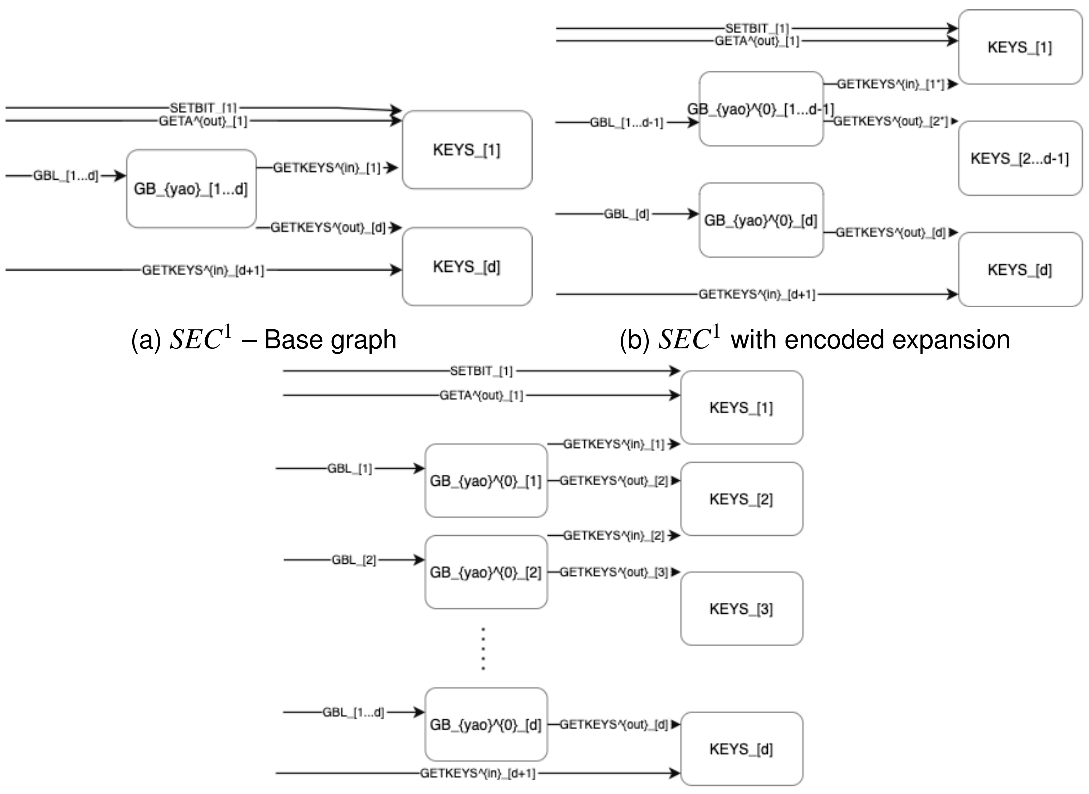
$$Adv(\mathcal{A}; \boldsymbol{SEC}^0(\boldsymbol{GB}), \boldsymbol{SEC}^1(\boldsymbol{SIM}))) \leq negl(n)$$

5.2.2 Required Graphs

In our evaluation we found that currently we could not to completely script from start to finish the main theorem (See p.14 in [30]) because of two main shortcomings of the:

1. We encounter edge ranges with arbitrary bases that are used to reason with the i^{th} expandable package. As previously mentioned in section 4.3.1), our expansion syntax doesn't currently support this.
2. The graphs involved in this proof require reducing to the security of multiple instances of the $KEYS$ package. Since the structure of our data format doesn't allow for different packages to share the same name (as we work in JSON), our substitution transformation therefore lacks the ability to generate the required graph.

Nevertheless, the tool is still useful in generating similar graphs that could be easily adapted to fulfil the needs of the user. Note that, since our tool suffers from package ordering issues, we performed extra package positioning and added "ghost" packages



(a) SEC^1 – Base graph (b) SEC^1 with encoded expansion

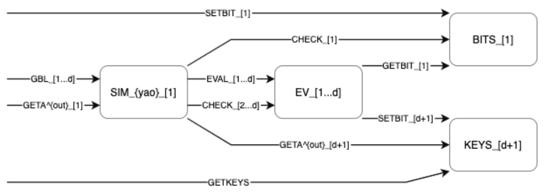
(c) SEC^1 expanded

Figure 5.3: The alternate forms of SEC^0

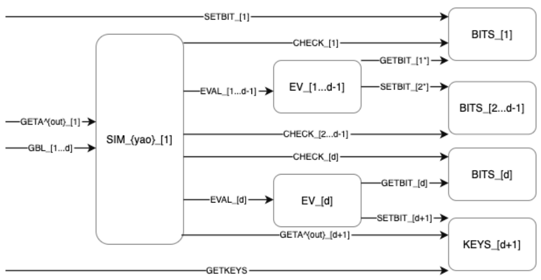
in Diagrams.net. We also include the package and name indexes as they exist in the tool to illustrate the function of expansion.

The alternate forms of SEC^0 (See p.11) are shown in Figure 5.3. The base graph we work from here is actually ambiguous in showing the relation of the $GETKEYS^{in}$ and $GETKEYS^{out}$ edges. Therefore we must first decompose GB_{yao} into $GB^0_{yao,1...d-1}$, $GB^0_{yao,d}$ and $KEYS_{2...d-1}$ with the appropriate encoding edges. We can then expand this package to a value of 2 with ghost edges to generate the required graphs. The transformation script used to generate these graphs can be found in Appendix D.1.

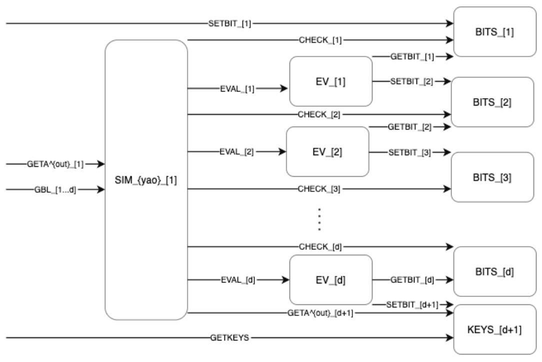
The alternate forms of SEC^1 (See p.12) are shown in Figure 5.4. Again the structure of this base graph is ambiguous therefore we encode within $EV_{1...d}$ an expandable package $EV_{1...d-1}$ and extra $BITS$ package with the necessary encoding edges. Once we have encoded this information the chain is expanded out to a value of 2, without ghost edges. We now decompose SIM_{yao} with an array of GB^1_{yao} and $AKEYS$ packages. To generate the subgraph required to do this, one can construct the graph shown in Appendix D.2 and expand it to a value of 2 without ghost edges. Once we have decomposed SIM_{yao}, the equivalence of $AKEYS$ and $BITS$ to $KEYS$ can be evaluated using substitution, thus generating the necessary forms of SEC^1. The transformation script and subgraphs used to generate these graphs can be found in Appendix D.3.



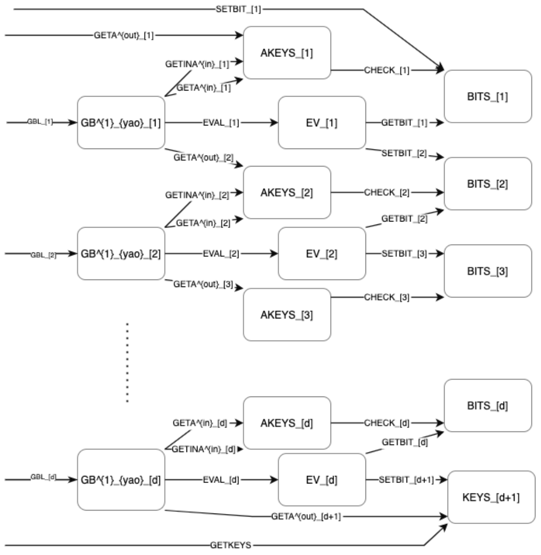
(a) SEC^0 – Base graph



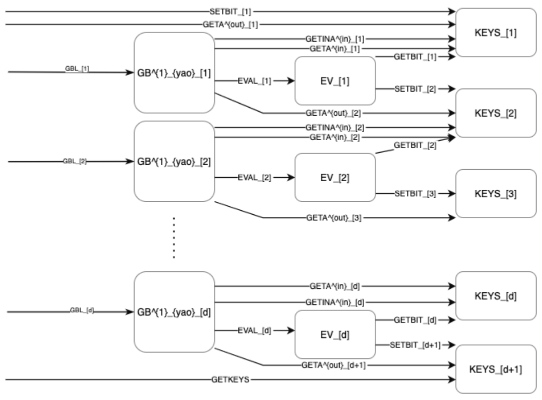
(b) SEC^0 with encoded expansion



(c) SEC^0 expanded



(d) SEC^0 with GB^1_{yao}



(e) SEC^0 with $AKEYS$ and $BITS$ replaced with $KEYS$

Figure 5.4: The alternate forms of SEC^1

5.3 Additional Features

5.3.1 Code Checking

Including package code verification for transformations would be a very valuable feature in this tool. This would not only further mitigate human error when generating graphs but could also open pathways for more accessible formal verification for those reviewing SSPs. This functionality was, however, not within scope of this project, but we envision that a future iteration of SSP-toolkit will integrate with such a mechanism within the **Packages** class.

5.3.2 Extras

There are a few other features we didn't manage to implement given the time constraints, the most valuable of which are listed below:

- A GUI for scriptable transformations. We imagine this as a sortable tree, much like the tree for modular packages.
- Making custom vertex targets for edges.
- Auto saving projects.
- Collaborative live project editing.
- An dedicated editor for package code.
- Mathematical typesetting on graphs.
- Customised reduction colours.
- Direct image and PDF export options.
- Advances styling features, such as ghost edges.

Chapter 6

Discussion and Conclusion

6.1 On automated proofs

An extension to this project was to build a tool that would serve as an automated proof finder. The general idea follows: A user would provide the structure of the base graph B and a target graph T. The user would also supply all the necessary assumptions (code equivalences) they think are necessary to get from B to T (as we work with *equivalences* this could also be thought of as a search from T to B). Then the tool would search over the possible graphs generated from performing a combination of these substitutions to automatically deduct if they are equivalent.

Automated proofs of this nature are not a *generative* application of graph rewriting but would rather require one to build a parser for graph *recognition*. This is a complex task as a recognition parser must search the most likely branches of candidate targets made from performing the supplied substitutions whilst being efficient enough to run in practice. Implementations of graph recognition parsers for context free grammars [32] have existed since 1989 – an exploration into modern graph recognition parsers is provided in this report by Blostein et al [33].

The first task in creating such a parser is defining how a transformation would be evaluated. A core feature of using SSPs is that we can decompose packages. This is especially advantageous when two packages don't share any *state*, therefore we will often see equivalences $L \rightarrow R$ in which either L or R is a disconnected graph. This makes equivalencies a bit ambiguous in our implementation of substitution. For example, assume L is disconnected and there are multiple complete occurrences of L, then there are multiple possibilities for an occurrence L to "match" together to replace with R. This increases the complexity of an SSP parser as each enumeration of these graphs (from L to R and vice versa) would need to be considered as a child of the base graph.

We hope that this insight will serve as a good starting point to whomever will build a specialised parser for SSPs.

6.2 Concluding remarks

SSP-toolkit is still in the prototyping phase. This is, in part, due to the methodology we adopted but also as a consequence of the extra functionality we added as the learning process for what really makes this tool useful, took place. There wasn't enough time to rigorously test each component therefore there are still critical bugs that we haven't pinned down yet. In our evaluation we displayed our tools usefulness by reducing the number of base graphs necessary to generate the required graphs for existing SSPs; However, we would've ideally provided an MVP for cryptographers to test and give qualitative feedback on. Even so, our evaluation still proved useful as we addressed some serious issues with the tool that will serve invaluable in the next iterations to come.

Bibliography

[1] C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss, "State separation for code-based game-playing proofs," *Lecture Notes in Computer Science Advances in Cryptology – ASIACRYPT 2018*, p. 222–249, 2018.

[2] A. W. Dent, "A note on game-hopping proofs," *IACR Cryptol. ePrint Arch.*, vol. 2006, p. 260, 2006.

[3] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[4] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, p. 120–126, Feb. 1978.

[5] G. Lowe, "An attack on the needham-schroeder public-key authentication protocol," *Information Processing Letters*, vol. 56, no. 3, pp. 131–133, 1995.

[6] A. Shamir, "A polynomial time algorithm for breaking the basic merkle-hellman cryptosystem," *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, 1982.

[7] C. E. Shannon, "Communication theory of secrecy systems," *The Bell System Technical Journal*, vol. 28, no. 4, pp. 656–715, 1949.

[8] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," *cryptology eprint archive*, 2004.

[9] J. Kilian and P. Rogaway, "How to protect des against exhaustive key search (an analysis of desx)," *Journal of Cryptology*, vol. 14, no. 1, pp. 17–35, 2001.

[10] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 409–426, Springer, 2006.

[11] G. Barthe, B. Grégoire, and S. Zanella Béguelin, "Formal certification of code-based cryptographic proofs," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 90–101, 2009.

[12] K. S. McCurley, "The discrete logarithm problem," in *Proc. of Symp. in Applied Math*, vol. 42, pp. 49–74, USA, 1990.

[13] I. Wegener, *Complexity Theory*. Springer-Verlag Berlin Heidelberg, 2005.

[14] J. Katz and Y. Lindell, *Introduction to modern cryptography*.

[15] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, 1971.

[16] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[17] C. McCreesh, P. Prosser, and J. Trimble, "The glasgow subgraph solver: using constraint programming to tackle hard subgraph isomorphism problem variants," in *International Conference on Graph Transformation*, pp. 316–324, Springer, 2020.

[18] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "Grgen: A fast spo-based graph rewriting tool," in *International Conference on Graph Transformation*, pp. 383–397, Springer, 2006.

[19] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[20] A. Lempel and I. Cederbaum, "Minimum feedback arc and vertex sets of a directed graph," *IEEE Transactions on circuit theory*, vol. 13, no. 4, pp. 399–403, 1966.

[21] P. Eades and L. Xuemin, "How to draw a directed graph," in *1989 IEEE Workshop on Visual Languages*, pp. 13–14, IEEE Computer Society, 1989.

[22] M. C. Newton, O. Sỳkora, M. Užovič, and I. Vrt'o, "New exact results and bounds for bipartite crossing numbers of meshes," in *International Symposium on Graph Drawing*, pp. 360–370, Springer, 2004.

[23] 2022.

[24] K. Puniamurthy, "A proof viewer for State-separating proofs: Yao's Garbling Scheme," master's thesis, Aalto University. School of Science, 2021.

[25] J. Shore and S. Warden, *The art of agile development*. " O'Reilly Media, Inc.", 2021.

[26] N. Völker, "Thoughts on requirements and design issues of user interfaces for proof assistants," *Electronic Notes in Theoretical Computer Science*, vol. 103, pp. 139–159, 2004.

[27] J. Herranz, D. Hofheinz, and E. Kiltz, "Some (in) sufficient conditions for secure hybrid encryption.," *Cryptology ePrint Archive*, 2006.

[28] A. C. Yao, "Protocols for secure computations," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, (Los Alamitos, CA, USA), pp. 160–164, IEEE Computer Society, nov 1982.

[29] A. C.-C. Yao, "How to generate and exchange secrets," in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pp. 162–167, 1986.

[30] C. Brzuska and S. Oechsner, "A state-separating proof for yao's garbling scheme," *Cryptology ePrint Archive*, 2021.

[31] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of garbled circuits," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 784–796, 2012.

[32] H. Bunke and B. Haller, "A parser for context free plex grammars," in *International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 136–150, Springer, 1989.

[33] D. Blostein, H. Fahmy, and A. Grbavec, "Practical use of graph rewriting," 1995.

Appendix A

File Formats

A.1 Project File Format

```
{
    "monolithic_pkgs" : {
        <package_name> : <package_code>
                            ...
    },
    "modular_pkgs" : {
        <graph_name> : {
            "oracles" : [[
                <edge_target>, <edge_name>
                            ...
            ]],
            "graph" : {
                <package_name> : [
                    <edge_target>, <edge_name>
                            ...
                ]
                    ...
            },
            "reduction" : ["<package_name>",...],
            "to_run" : {
                ...
            },
            "history" : {
                ...
            }
        }
    }
}
```

A.2 Transformation Script Format

```
"to_run": {
    "Expanded Graph" : {
        "expandable_package" : "<package_name>",
        "value" : <value>             // Integer
        "ghost" : true || false       // Optional
        "to_run" : {...}              // Optional
    },
    "Decomposed Graph" : {
        "target" : "<package_name>",
        "subgraph" : "<graph_name>" || {<graph_data>},
        "to_run" : {...}              // Optional
    },
    "Composed Graph" : {
        "packages" : ["<package_name>",...],
        "new_package_name" : "<new_package_name>"
        "to_run" : {...}              // Optional
    },
    "Substituted Graph" : {
        "lhs" : "<graph_name>" || {<graph_data>},
        "rhs" : "<graph_name>" || {<graph_data>},
        "partial" : true || false     // Optional
        "to_run" : {...}              // Optional
    },
    "Reduced Graph" : {
        "reduction" : ["<package_name>",...],
        "bitstring" : "<bit_string>"
        "to_run" : {...}              // Optional
    }
    ...
}
```

Appendix B

Expansion Pseudocode

Algorithm 3 findAllChains, **Input:** G

```
1: $chains = \{\}$
2: $visited = \{\}$
3: for $v \in G.V$ do
4:     if $'...' \in v.name$ and $v \notin visited$ then
5:         $chains.add(findChain(v, visited, \perp))$
6:     end if
7: end for
8: return $chains$
```

Appendix C

KEM-DEM

C.1 KEM-DEM Package Code

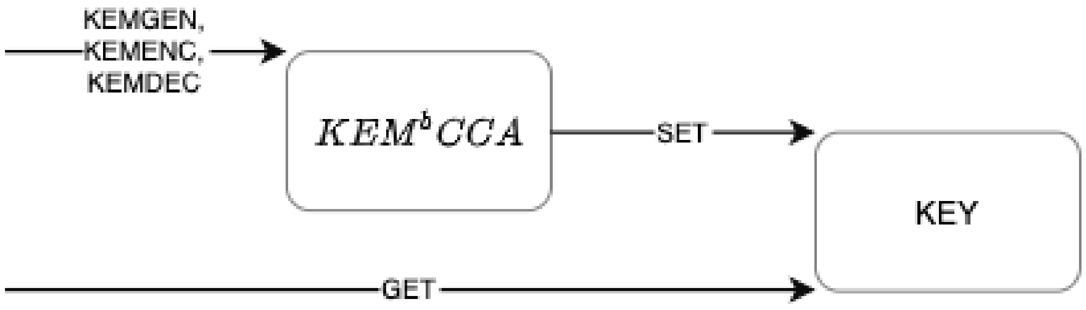
```
KEY
GEN()
  assert $k = \bot$
  $k \leftarrow\!\$ \{0,1\}^n$

GET()
  assert $k \neq \bot$
  return $k$

SET($k'$)
  assert $k \neq \bot$
  $k = k'$
```

```
MOD-CCA$^b$
PKGEN()
  assert $sk = \bot$
  $pk \leftarrow KEMGEN()$
  return $pk$

PKENC(m)
  assert $pk \neq \bot$
  assert $c = \bot$
  $c_1 \leftarrow ENCAP()$
  $c_2 \leftarrow ENC(m)$
  $c \leftarrow c_1 || c_2$
  return $c$

PKDEC(c')
  assert $pk \neq \bot$
  assert $c' \neq c$
  $c'_1 || c'_2 \leftarrow c'$
  if $c'_1 = c_1$ then
    $m \leftarrow DEC(c'_2$
  else
    $k' \leftarrow DECAP(c'_1)$
    $m \leftarrow \theta.dec(k', c'_2)$
  return $m$
```

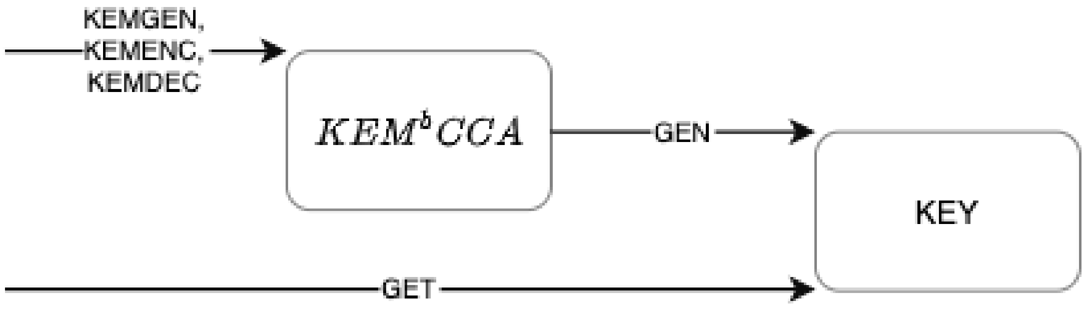
```
KEM$^b$
KEMGEN()
  assert $sk = \bot$
  $pk, sk \leftarrow\!\$ \eta.kgen()$
  return $pk$

ENCAP()
  assert $pk \neq \bot$
  assert $c = \bot$
  if $b = 0$ then
    $k, c \leftarrow\!\$ \eta.encap(pk)$
    $SET(k)$
  else
    $k, c \leftarrow\!\$ \eta.encap(pk)$
    $GEN()$
  return $c$

DECAP(c')
  assert $sk \neq \bot$
  assert $c' \neq c$
  $k \leftarrow\!\$ \eta.decap(sk, c')$
  return $k$
```

```
DEM$^b$
ENC(m)
  $c \neq \bot$
  $k \leftarrow GET()$
  if $b = 0$ then
    $c \leftarrow \theta.enc(k, m)$
  else
    $c \leftarrow \theta.enc(k, 0^{|m|})$
  return $c$

DEC(c')
  assert $c \neq c'$
  $k \leftarrow GET()$
  $m \leftarrow \theta.dec(k, c')$
  return $m$
```

C.2 Transformation Input



(a) **KEM_assumption_L**



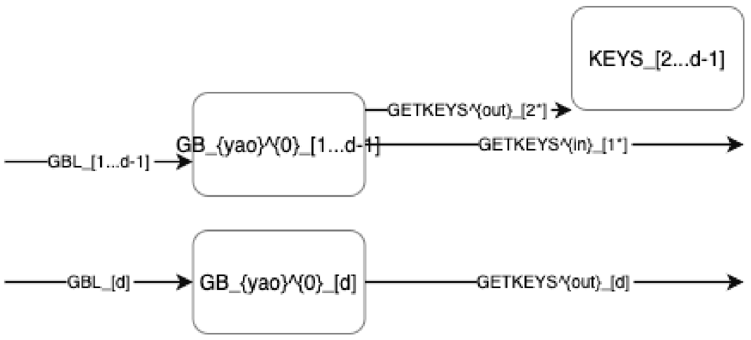
(b) **KEM_assumption_R**

```
    "Graph_B_1": {
      "type": "reduce",
      "reduction": [
        "MOD_CCA_[1]",
        "DEM^{0}_CCA_[1]"
      ],
      "bitstring": "b",
      "to_run": {
        "Graph_B_2": {
          "type": "substitute",
          "lhs": "KEM_assumption_L",
          "rhs": "KEM_assumption_R",
          "partial": true,
          "to_run": {
            "Graph_C": {
              "type": "reduce",
              "reduction": [],
              "bitstring": "1",
              "to_run": {
                "Graph_D": {
                  "type": "reduce",
                  "reduction": [
                    "MOD_CCA_[1]",
                    "KEM^{1}_CCA_[1]"
                  ],
                  "bitstring": "b",
                  "to_run": {
                    "Graph_E": {
                      "type": "reduce",
                      "reduction": [],
                      "bitstring": "1",
                      "to_run": {
                        "Graph_F_1": {
                          "type": "reduce",
                          "reduction": [
                            "MOD_CCA_[1]",
                            "DEM^{1}_CCA_[1]"
                          ],
                          "bitstring": "b",
                          "to_run": {
                            "Graph_F_2": {
                              "type": "substitute",
                              "lhs": "KEM_assumption_R",
                              "rhs": "KEM_assumption_L",
                              "partial": true,
                              "to_run": {
                                "Graph_G": {
                                  "type": "reduce",
                                  "reduction": [],
                                  "bitstring": "0"
                                }
                ...
    }
```

Appendix D

Yao's Garbled Circuit

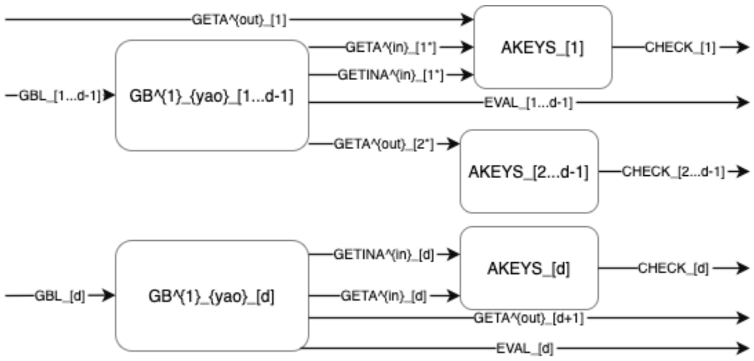
D.1 SEC_0 Transformation Input



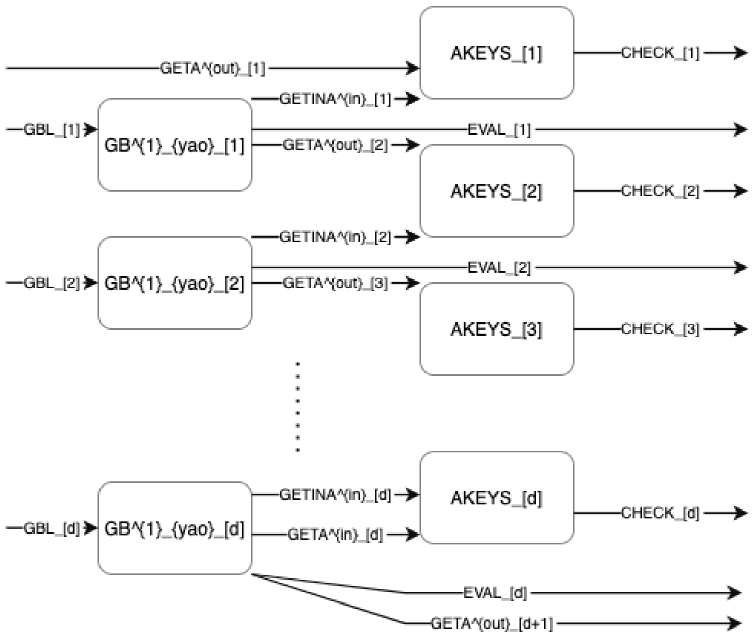
(a) **SEC0_Encoding - SUBGRAPH**

```
"SEC0_C_1": {
  "type": "decompose",
  "target": "GB_{yao}_[1...d]",
  "subgraph": "SEC0_Encoding - SUBGRAPH",
  "to_run": {
    "SEC0_C": {
      "type": "expand",
      "expandable_package": "GB_{yao}^{0}_[1...d-1]",
      "value": 2,
      "ghost": true
    }
  }
}
```

D.2 SIM_{yao} Transformation Input



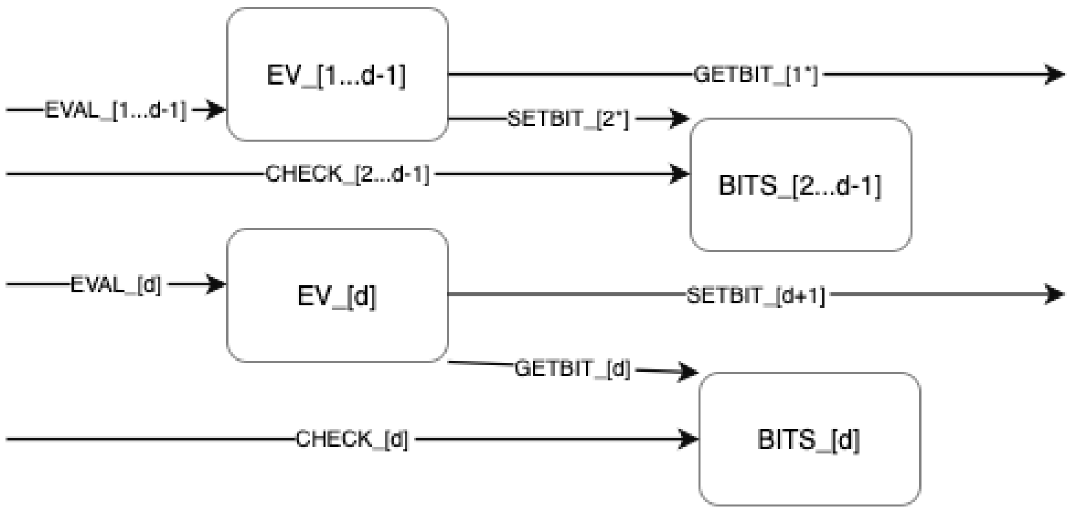
(a) Base graph



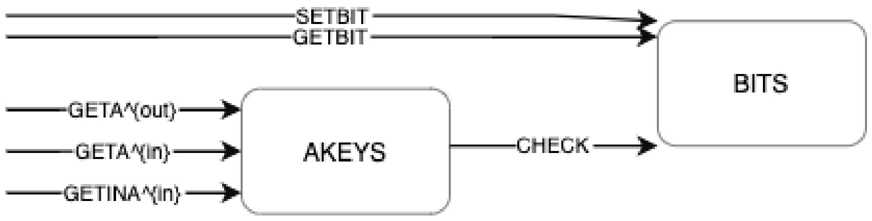
(b) **GB1_sub**

```
            "GB1_sub": {
                "type": "expand",
                "expandable_package": "GB^{1}_{
                  yao}_[1...d-1]",
                "value": 2,
                "ghost": false
            }
```

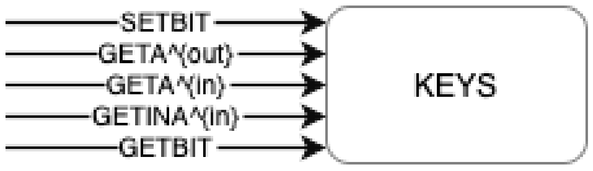
D.3 SEC_1 Transformation Input



(a) **SEC1_Encoding - SUBGRAPH**



(b) **AKEYS_L**



(c) **KEYS_R**

```
"SEC1_B_1": {
    "type": "decompose",
    "target": "EV_[1...d]",
    "subgraph": "SEC1_Encoding - SUBGRAPH",
    "to_run": {
      "SEC1_B": {
        "type": "expand",
        "expandable_package": "EV_[1...d-1]",
        "value": 2,
        "ghost": false,
        "to_run": {
          "SEC1_C": {
            "type": "decompose",
            "target": "SIM_{yao}_[1]",
            "subgraph": "GB1_sub",
            "to_run": {
              "SEC1_D": {
                "type": "substitute",
                "lhs": "AKEYS_L",
                "rhs": "KEYS_R",
                "partial": true
              }
        ...
}
```

Appendix E

Issues with MxGraph

Below is a short list of the issues faced whilst working with MxGraph, these are notable as they slowed down development majorly impacted the time taken to implement the features of the tool. It's very likely that this is because the tool uses Node.js to provide libraries server-side whilst MxGraph is meant to be included as a client-side library, thus many workarounds had to be put in place because of this. It's important to get across that MxGraph is an *excellent* library and JGraph are very generous in making it open source, however integrating the library in a Node.js project was just a very tedious and experimental process.

- The MxEditor class contains all the features that a great graph editor needs, however the documentation on how to implement it is very ambiguous and therefore we weren't able to utilise it and had to manually add all the features listed in section 4.7.2.
- Some coordinates of an MxGraph cell are flipped on the y axis for no apparent reason.
- The coordinates of vertex cells are defined with respect to their parent cell which we can set to be the same for all vertices in a graph, however, an edges exit and entry points and their virtual bends coordinates are defined with respect to the source or target node making defining absolute points for use in geometric algorithms a very convoluted process.
- There are no examples by JGraph for integration with Node.js; I owe my life to the MxGraph contributors on stack overflow.